A grammar based approach towards the automatic implementation of data communication protocols in hardware
Bloks, R.H.J.

DOI: 10.6100/IR406597
Published: 01/01/1993

Citation for published version (APA):
A Grammar Based Approach towards the Automatic Implementation of Data Communication Protocols in Hardware

R.H.J. Bloks
A Grammar Based Approach towards the Automatic Implementation of Data Communication Protocols in Hardware

PROEFSCHRIFT

ter verkrijging van de graad van doctor aan de Technische Universiteit Eindhoven, op gezag van de Rector Magnificus, prof.dr. J.H. van Lint, voor een commissie aangewezen door het College van Dekanen in het openbaar te verdedigen op vrijdag 10 december 1993 om 16.00 uur

door

Rudolf Henricus Johannes Bloks

geboren te Eindhoven
Dit proefschrift is goedgekeurd door de promotoren:

prof.ir. M.P.J. Stevens
prof.Dr.Ing. J.A.G. Jess

CIP-DATA KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Bloks, Rudolf Henricus Johannes

A grammar based approach towards the automatic implementation of data communication protocols in hardware / Rudolf Henricus Johannes Bloks. - [S.l. : s.n.]. - Fig., tab.
ISBN 90-9006548-2
NUGI 832/853
Subject headings: data communication protocols / specification languages.
The research work presented in this thesis was carried out at the Digital Information Systems Group, Faculty of Electrical Engineering of the Eindhoven University of Technology, on the automatic hardware implementation of data communication protocols from formal descriptions.

The results of the work I present are based on a number of different disciplines, such as abstract mathematics for the formal foundation, digital electronics, designing and implementing complex circuits for the target architecture, and computer programming to construct the necessary software tools.

Although this protocol engine project has always been a one-man project, I had help from a few students in the design and simulation of the target architecture as well as in the creation of a test case. It would have been impossible to do everything myself and to get the same results. Even now while I am writing this thesis, the project is not yet finished, but fortunately I can say that the project will be continued and hopefully completed within the next two years.

• • • •

I would like to express my gratitude to all people who have helped me through this project and with the completion of this thesis. I want to thank IBM for sponsoring our group, thereby enabling this project to be started in the first place. There are many people involved, but some of them I would like to thank especially: prof. ir. M. P. J. Stevens for giving me the opportunity to do a Ph.D. research on a very interesting topic, ir. W. Rovers and ir. J. Voeten for reading and suggesting corrections for the first draft copy of this thesis and together with ir. S. El Kassas for the many interesting discussions on all kinds of topics including of course the one of this thesis, dr. ir. A. Verschueren for providing a design and simulation tool, ir. F. Budzelaar for his support in programming PC’s, ir. J. v. Lunteren, M. Jacobs, L. Geurts and A. Brouwer for their active participation in the design and testing of the target architecture and a test case, and all people of the Digital Information Systems group who have made my stay there unforgettable. Finally, I would like to thank my family for supporting me in my study.

Eindhoven, May 1993

- iii -
Summary

Exchange of data between computers over networks is only possible when the equipment involved observes the rules of some specified protocol. Such protocols are usually very complex and nearly all implementations are done in software running on the processing units involved. A disadvantage of software implementations is that they are relatively slow when compared to hardware solutions. Due to the increasing demand for faster data exchange and the fact that network technology can now provide much higher bandwidths than software implementations of protocols can utilize, new kinds of implementations must be found, preferably in hardware.

To reduce cost, turn around time and error risk, implementations should be generated automatically. This requires a formal language in which a protocol implementation can be described, and from which a hardware architecture can be derived algorithmically. The definition of such a language and a method for generating an implementation from it form the major part of this Ph.D. project.

This thesis describes one possible approach to generate hardware implementations automatically from formal high level descriptions. It is based on the idea of considering a protocol to be a specification of a symbolic language consisting of input and output actions. The allowed sentences are defined by rules of the grammar for that language. The concept of formal languages, grammars and related automata has been developed to a considerable extent during the last few decades. In this thesis, the general context-free grammar is taken as a basis and extended with attributes (used to store protocol variables and other context information), bidirectional communication (input as well as output symbols) and conditional rules (to allow context dependent parsing) for very complex behaviour descriptions, resulting in a protocol grammar that can be used to describe the symbolic protocol language. Similarly, the pushdown automaton which can be used to implement any context free grammar is extended with attribute management and a conditional parsing mechanism to obtain an abstract implementation for protocol grammars called a protocol pushdown automaton. After formally defining the concept of an accepted/generated language for both the protocol grammar and the protocol pushdown automaton, it is proven that any protocol grammar can indeed be implemented in a protocol pushdown automaton and an algorithm is given for its construction.
To obtain a physical implementation for the protocol pushdown automaton, its rather abstract operations were mathematically transformed into concrete implementable ones. This finally resulted in the grammar processor, a deterministic finite version of the protocol pushdown automaton which has been designed and tested in a simulation environment and which forms the key concept in the hardware architecture of protocol implementations.

Protocol engines consist of networks of interconnected grammar processors, each implementing a part of the entire protocol (e.g. a layer or part of a layer). This subdivision can be chosen arbitrarily. The actual data to be exchanged is stored and processed in a separate shared packet buffer memory whose management functionality is also specified in this thesis and fully implementable in hardware. To obtain high throughput (over 250 thousand packets per second) special memory management algorithms were invented. A start has been made with its implementation.

A step is made towards setting up a complete design system for protocol engines based on this technique. This system will ultimately contain all tools necessary to create an implementation of a protocol. The grammar compiler is finished and the hardware linker will be completed in some time. A high level specification language (LOTOS) to protocol grammars converter would be an interesting extension, as well as a complete performance analysis tool and a software implementation generator. When finished, the combined tools will provide the user with a system that enables him/her to make implementations of protocols very rapidly in hardware and perhaps in software as well. In this thesis, ideas are given for the further implementation of unfinished parts.

Finally, a simple stochastic model for protocol grammars is developed, by which it is possible to make estimates of achievable performance when protocols are implemented using the grammar processor. Application to an X.25 test case shows that extremely high performance is possible.
Overview of the Chapters

Chapter 1 introduces the concept of protocols and discusses some important protocol related issues, in particular the implementation problem. It is argued that in order to keep up with network technology and bandwidth demands, hardware implementations will become necessary, and that these should be generated automatically from formal protocol descriptions.

To achieve automatic implementation, a description must be given in a formal language. Hence, a suitable description language must be found. Chapter 2 presents a list of requirements for formal protocol specification languages and shows how formal grammars can be used to describe protocol implementations.

Chapter 3 starts by introducing standard grammars and their hierarchy. Context-free grammars are chosen as a basis for the language. They are first extended to attribute grammars and subsequently to protocol grammars. A formal definition of the accepted/generated language of a protocol grammar is given using leftmost derivations. Finally, it is shown how protocol grammars fit in the OSI protocol model.

Chapter 4 extends the standard pushdown automaton which can be used to implement context-free grammars. The result is a protocol pushdown automaton, which can be programmed to implement any protocol grammar. The concept of the accepted/generated language is formally defined, and an algorithm is given for the construction of a protocol pushdown automaton from a protocol grammar such that both accept/generate the same language. This algorithm is mathematically proven. Finally, some issues concerning finiteness and nondeterminism of the automaton are discussed, as well as its relation to protocol engines.

Chapter 5 describes a possible implementation of the protocol pushdown automaton. It is shown how the highly abstract operations can be transformed into concrete implementable ones. This leads to a design (called a grammar processor) which can in principle be implemented on a microchip using current technology.

Chapter 6 shows the structure of a complete protocol engine, consisting of a number of cooperating grammar processors and a packet management system. Specifications are given for the operation of the memory and packet management. Furthermore, communication channels, events and error handling are introduced.
Chapter 7 presents the entire design system that chapter 1 introduced as the goal of this research project. It shows how protocols are first described in a set of protocol grammars, then compiled into code and tables, and finally implemented in a custom generated protocol engine as given in chapter 6.

In chapter 8, a simple model is given by which a quick performance analysis can be done of any protocol implementation obtained in this way. It is based on a stochastic analysis of the basic context-free grammar underlying any protocol grammar. As an example, the model is applied to an X.25 test case.

Finally, chapter 9 presents the conclusions and overall results of this work, as well as a number of recommendations for future continuation of this project.
## Table of Contents

### List of Symbols .............................................. xv

**Chapter 1**

1.1 General Introduction ..................................... 1

1.1.1 Data communication related problems ......... 1

1.1.2 Transmission errors .................................. 1

1.1.3 Medium access control .............................. 2

1.2 Communication protocols ............................... 2

1.2.1 Informal definition .................................. 2

1.2.2 Hardware and software implementations ....... 2

1.2.3 Protocol classes ...................................... 3

1.2.3.1 Character-oriented protocols ................. 3

1.2.3.2 Bit-oriented protocols ........................ 4

1.2.4 Protocol standardization ........................... 4

1.2.5 Concepts of the OSI reference model .......... 5

1.2.5.1 Layered architecture ......................... 5

1.2.5.2 Naming conventions ........................... 5

1.2.5.3 Service primitives ........................... 7

1.2.5.4 Service Data Units and Protocol Data Units ... 7

1.3 Protocol Implementations ............................... 8

1.3.1 Hardware versus software implementations .... 8

1.3.2 The software bottleneck ......................... 9

1.3.3 Other research, work and products for protocol implementations ...... 10

1.4 Protocol engine concepts ............................... 12

1.5 Research aim ........................................... 14

1.5.1 A protocol engine design system ............... 14

1.5.2 Research areas within the project ............... 15

**Chapter 2**

2.1 Formal Protocol Specification Languages ............ 17

2.1.1 Protocol development ............................. 17

2.1.2 Correctness of protocols ....................... 18

2.1.3 Protocol layers .................................... 19

2.1.4 Design trajectory .................................. 20

2.2 Properties of protocol specification languages .... 20

2.3 Formal Grammars ....................................... 21

2.3.1 Automatic implementation ...................... 23

2.3.2 Verification and testing with grammars ....... 24

2.4 Conclusion ............................................. 24

**Chapter 3**

3.1 Protocol Grammars ....................................... 25

3.1.1 Introduction ....................................... 26

3.2 The Chomsky hierarchy ................................ 26

3.2.1 Unrestricted (type 0) grammars ............... 26

3.2.2 Context-sensitive (type 1) grammars .......... 27
Table of Contents (continued)

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.3</td>
<td>Context-free (type 2) grammars</td>
<td>28</td>
</tr>
<tr>
<td>3.2.4</td>
<td>Regular (type 3) grammars</td>
<td>29</td>
</tr>
<tr>
<td>3.3</td>
<td>Derivation trees</td>
<td>30</td>
</tr>
<tr>
<td>3.4</td>
<td>Context free grammars and pushdown automata</td>
<td>30</td>
</tr>
<tr>
<td>3.5</td>
<td>Protocols and the expressive power of grammars</td>
<td>32</td>
</tr>
<tr>
<td>3.5.1</td>
<td>Separation of data and control space</td>
<td>33</td>
</tr>
<tr>
<td>3.5.2</td>
<td>Expression of unbounded recursion</td>
<td>33</td>
</tr>
<tr>
<td>3.5.3</td>
<td>Unbounded state systems</td>
<td>34</td>
</tr>
<tr>
<td>3.5.4</td>
<td>Conclusion</td>
<td>36</td>
</tr>
<tr>
<td>3.6</td>
<td>Attributed context free grammars</td>
<td>36</td>
</tr>
<tr>
<td>3.7</td>
<td>Protocol grammars</td>
<td>42</td>
</tr>
<tr>
<td>3.8</td>
<td>Example of a protocol grammar</td>
<td>46</td>
</tr>
<tr>
<td>3.9</td>
<td>Derivability and language definition</td>
<td>48</td>
</tr>
<tr>
<td>3.9.1</td>
<td>Operations and data types</td>
<td>48</td>
</tr>
<tr>
<td>3.9.2</td>
<td>Expression computability</td>
<td>50</td>
</tr>
<tr>
<td>3.9.3</td>
<td>A mathematical reshape of the production rules</td>
<td>51</td>
</tr>
<tr>
<td>3.9.4</td>
<td>Introduction of endmarkers</td>
<td>52</td>
</tr>
<tr>
<td>3.9.5</td>
<td>Leftmost derivation steps</td>
<td>53</td>
</tr>
<tr>
<td>3.9.6</td>
<td>Language definition</td>
<td>56</td>
</tr>
<tr>
<td>3.10</td>
<td>Modeling of layer hierarchy</td>
<td>57</td>
</tr>
<tr>
<td>3.10.1</td>
<td>Layers, entities and services</td>
<td>57</td>
</tr>
<tr>
<td>3.10.2</td>
<td>Connected grammars</td>
<td>58</td>
</tr>
<tr>
<td>3.10.2.1</td>
<td>Hierarchical connections</td>
<td>58</td>
</tr>
<tr>
<td>3.10.2.2</td>
<td>Non-hierarchical connections</td>
<td>58</td>
</tr>
</tbody>
</table>

Chapter 4  Protocol Automata

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Pushdown automata</td>
<td>61</td>
</tr>
<tr>
<td>4.2</td>
<td>Extensions to the pushdown automaton</td>
<td>62</td>
</tr>
<tr>
<td>4.3</td>
<td>Attribute storage management</td>
<td>64</td>
</tr>
<tr>
<td>4.4</td>
<td>Endmarkers</td>
<td>66</td>
</tr>
<tr>
<td>4.5</td>
<td>The formal protocol pushdown automaton</td>
<td>67</td>
</tr>
<tr>
<td>4.5.1</td>
<td>String derivability and language definition</td>
<td>68</td>
</tr>
<tr>
<td>4.5.2</td>
<td>Construction algorithm for the automaton</td>
<td>68</td>
</tr>
<tr>
<td>4.5.3</td>
<td>Proof of the construction</td>
<td>72</td>
</tr>
<tr>
<td>4.6</td>
<td>Problems regarding physical implementations</td>
<td>74</td>
</tr>
<tr>
<td>4.6.1</td>
<td>The stability property of protocols</td>
<td>78</td>
</tr>
<tr>
<td>4.6.2</td>
<td>Protocol grammars and stability</td>
<td>79</td>
</tr>
<tr>
<td>4.6.2.1</td>
<td>Endless loop constructs</td>
<td>80</td>
</tr>
<tr>
<td>4.6.2.2</td>
<td>Limiting right recursion in protocol grammars</td>
<td>80</td>
</tr>
<tr>
<td>4.6.2.3</td>
<td>Stable construction of endless loops</td>
<td>81</td>
</tr>
<tr>
<td>4.6.3</td>
<td>Elimination of nondeterminism</td>
<td>81</td>
</tr>
<tr>
<td>4.6.3.1</td>
<td>Deterministic parsing strategies</td>
<td>82</td>
</tr>
<tr>
<td>4.6.3.2</td>
<td>LR parsing techniques</td>
<td>83</td>
</tr>
<tr>
<td>4.6.3.3</td>
<td>Predictive parsing techniques</td>
<td>84</td>
</tr>
<tr>
<td>4.6.4</td>
<td>Conclusions concerning implementations</td>
<td>85</td>
</tr>
<tr>
<td>Chapter 5</td>
<td>A Physical Implementation of a Protocol Automaton</td>
<td>87</td>
</tr>
<tr>
<td>----------</td>
<td>-----------------------------------------------</td>
<td>----</td>
</tr>
<tr>
<td>5.1</td>
<td>Functionality of the protocol pushdown automaton</td>
<td>88</td>
</tr>
<tr>
<td>5.2</td>
<td>Conversion of abstract operations</td>
<td>89</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Environments</td>
<td>89</td>
</tr>
<tr>
<td>5.2.2</td>
<td>Attribute expressions and action procedures</td>
<td>90</td>
</tr>
<tr>
<td>5.2.3</td>
<td>Nonterminal symbol processing</td>
<td>91</td>
</tr>
<tr>
<td>5.2.3.1</td>
<td>Attribute passing mechanism</td>
<td>92</td>
</tr>
<tr>
<td>5.2.3.2</td>
<td>Local environment allocation</td>
<td>93</td>
</tr>
<tr>
<td>5.2.3.3</td>
<td>Enable conditions and rule selection</td>
<td>94</td>
</tr>
<tr>
<td>5.2.4</td>
<td>Environment restores for endmarkers</td>
<td>95</td>
</tr>
<tr>
<td>5.2.5</td>
<td>Attribute access, allocation and deallocation</td>
<td>96</td>
</tr>
<tr>
<td>5.2.6</td>
<td>Specification of the attribute operator</td>
<td>96</td>
</tr>
<tr>
<td>5.3</td>
<td>The grammar processor</td>
<td>98</td>
</tr>
<tr>
<td>5.4</td>
<td>The parse stack</td>
<td>99</td>
</tr>
<tr>
<td>5.5</td>
<td>Invocation of action procedures</td>
<td>100</td>
</tr>
<tr>
<td>5.6</td>
<td>Enable conditions for production rules</td>
<td>101</td>
</tr>
<tr>
<td>5.7</td>
<td>Input and output operations</td>
<td>102</td>
</tr>
<tr>
<td>5.8</td>
<td>The Pushdown Controller</td>
<td>102</td>
</tr>
<tr>
<td>5.9</td>
<td>The Attribute Evaluator</td>
<td>105</td>
</tr>
<tr>
<td>5.9.1</td>
<td>The global architecture</td>
<td>105</td>
</tr>
<tr>
<td>5.9.2</td>
<td>Instruction execution and pipelining</td>
<td>106</td>
</tr>
<tr>
<td>5.9.3</td>
<td>Data types and operations</td>
<td>107</td>
</tr>
<tr>
<td>5.10</td>
<td>Memory usage of local environments</td>
<td>108</td>
</tr>
<tr>
<td>5.11</td>
<td>Attribute allocation map</td>
<td>109</td>
</tr>
<tr>
<td>5.12</td>
<td>Conclusion</td>
<td>112</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Protocol Engines</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>Packet management system</td>
<td>113</td>
</tr>
<tr>
<td>6.1.1</td>
<td>Packet storage and transfer</td>
<td>114</td>
</tr>
<tr>
<td>6.1.2</td>
<td>Requirements for the packet management system</td>
<td>116</td>
</tr>
<tr>
<td>6.1.3</td>
<td>Memory management characteristics</td>
<td>119</td>
</tr>
<tr>
<td>6.1.4</td>
<td>Primitive functions of the packet management system</td>
<td>120</td>
</tr>
<tr>
<td>6.1.5</td>
<td>Specification of the memory manager</td>
<td>121</td>
</tr>
<tr>
<td>6.1.6</td>
<td>Specification of the packet manager</td>
<td>125</td>
</tr>
<tr>
<td>6.1.7</td>
<td>The use of packet references</td>
<td>129</td>
</tr>
<tr>
<td>6.1.8</td>
<td>Packet manager interfaces</td>
<td>130</td>
</tr>
<tr>
<td>6.1.8.1</td>
<td>Grammar processor command interface</td>
<td>131</td>
</tr>
<tr>
<td>6.1.8.2</td>
<td>Host and DCE interface</td>
<td>132</td>
</tr>
<tr>
<td>6.1.8.3</td>
<td>Packet assemblers and disassemblers</td>
<td>132</td>
</tr>
<tr>
<td>6.2</td>
<td>Communication channels</td>
<td>133</td>
</tr>
<tr>
<td>6.2.1</td>
<td>Functionality</td>
<td>133</td>
</tr>
<tr>
<td>6.2.2</td>
<td>Channel types</td>
<td>133</td>
</tr>
<tr>
<td>6.2.3</td>
<td>Message translation</td>
<td>134</td>
</tr>
</tbody>
</table>
## Table of Contents (continued)

6.3 Events and errors .................................................. 135
  6.3.1 Exceptional situations ........................................... 135
  6.3.2 Exception handling strategies .................................... 136
  6.3.3 Semantics of input and standard events ......................... 138
  6.3.4 Using errors and events .......................................... 140
6.4 An architecture for protocol engines ................................ 140
6.5 Conclusion .................................................................. 142

### Chapter 7

A Design System for Protocol Engines .................................. 143
  7.1 The design path ...................................................... 143
  7.2 A specification language for protocol grammars ................... 146
  7.3 The module library and hardware compiler .......................... 146
  7.3.1 Library management ............................................... 147
  7.3.2 Generation of silicon implementations .......................... 147
  7.4 Language and implementation aspects ............................... 148

### Chapter 8

A Stochastic Performance Analysis Model ............................... 149
  8.1 Concepts of the model ............................................... 150
  8.2 Probabilistic context-free grammars ................................ 151
  8.3 Computation of mean nonterminal reduce time ...................... 152
  8.4 Computation of mean terminal count ................................ 154
  8.5 Estimation of the rule search time .................................. 155
  8.6 Endless protocol systems ............................................ 156
  8.7 The X.25 protocol: an example ...................................... 158
  8.8 Conclusion .................................................................. 164

### Chapter 9

Conclusions and Directions for Future Work ............................ 165
  9.1 History and status of the research project ......................... 165
  9.2 Conclusions from the research results ............................... 166
  9.3 Overall conclusions ................................................... 168
  9.4 Recommendations for future work .................................... 168

References ........................................................................... 171

Samenvatting ........................................................................ 177

Curriculum Vitae .................................................................... 179
# List of Figures and Tables

<table>
<thead>
<tr>
<th>Chapter 1</th>
<th>General Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1.1</td>
<td>The OSI communications model.</td>
<td>5</td>
</tr>
<tr>
<td>Figure 1.2</td>
<td>Layer entities and Service Access Points.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.3</td>
<td>Connection endpoints and SAP connections.</td>
<td>6</td>
</tr>
<tr>
<td>Figure 1.4</td>
<td>Naming of data units in layers.</td>
<td>7</td>
</tr>
<tr>
<td>Figure 1.5</td>
<td>PDUs are sent between peer entities and SDUs between adjacent layers.</td>
<td>8</td>
</tr>
<tr>
<td>Figure 1.6</td>
<td>The position of the protocol engine in a communication system.</td>
<td>12</td>
</tr>
<tr>
<td>Figure 1.7</td>
<td>Partitioning of the protocol implementation.</td>
<td>13</td>
</tr>
<tr>
<td>Figure 1.8</td>
<td>A front end system for protocols using HDLC.</td>
<td>13</td>
</tr>
</tbody>
</table>

| Chapter 2 | Formal Protocol Specification Languages | 17 |

<table>
<thead>
<tr>
<th>Chapter 3</th>
<th>Protocol Grammars</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 3.1</td>
<td>Pushdown Automaton.</td>
<td>31</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>The FSM for the PDA that recognizes $a^n b^n (n \geq 1)$</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Implementation of L in a finite state machine.</td>
<td>35</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 4</th>
<th>Protocol Automata</th>
<th>61</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 4.1</td>
<td>Pushdown Automaton.</td>
<td>62</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>The Protocol Pushdown Automaton (PPDA).</td>
<td>65</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 5</th>
<th>A Physical Implementation of a Protocol Automaton</th>
<th>87</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 5.1</td>
<td>The architecture for the grammar processor</td>
<td>98</td>
</tr>
<tr>
<td>Figure 5.2</td>
<td>The structure of the parse stack.</td>
<td>100</td>
</tr>
<tr>
<td>Figure 5.3</td>
<td>Internal architecture of the Pushdown Controller.</td>
<td>103</td>
</tr>
<tr>
<td>Figure 5.4</td>
<td>The Attribute Evaluator.</td>
<td>106</td>
</tr>
<tr>
<td>Figure 5.5</td>
<td>Memory map of the attribute RAM.</td>
<td>110</td>
</tr>
<tr>
<td>Figure 5.6</td>
<td>Memory Allocation for new rules.</td>
<td>111</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 6</th>
<th>Protocol Engines</th>
<th>113</th>
</tr>
</thead>
<tbody>
<tr>
<td>Table 6.1</td>
<td>Advantages and disadvantages of global and local packet memories</td>
<td>115</td>
</tr>
<tr>
<td>Figure 6.1</td>
<td>Packet operations for each layer from CCITT Rec. X.200 (1993).</td>
<td>118</td>
</tr>
<tr>
<td>Table 6.2</td>
<td>Packet length conversion operations indicated in figure 6.1.</td>
<td>118</td>
</tr>
<tr>
<td>Figure 6.2</td>
<td>Layered structure of the packet management system.</td>
<td>121</td>
</tr>
<tr>
<td>Figure 6.3</td>
<td>Users of the packet management system and their interconnections.</td>
<td>130</td>
</tr>
<tr>
<td>Figure 6.4</td>
<td>Structure of a single receiver stage.</td>
<td>141</td>
</tr>
<tr>
<td>Figure 6.5</td>
<td>Structure of a single transmitter stage.</td>
<td>141</td>
</tr>
<tr>
<td>Figure 6.6</td>
<td>High level protocol engine architecture (layers N and N+1).</td>
<td>142</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Chapter 7</th>
<th>A Design System for Protocol Engines</th>
<th>143</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 7.1</td>
<td>General protocol design and implementation trajectory.</td>
<td>144</td>
</tr>
<tr>
<td>Figure 7.2</td>
<td>Overview of the design system.</td>
<td>145</td>
</tr>
<tr>
<td>Figure 7.3</td>
<td>Languages, grammars and automata involved in the design system.</td>
<td>148</td>
</tr>
</tbody>
</table>
## List of Figures and Tables (continued)

<table>
<thead>
<tr>
<th>Chapter 8</th>
<th>A Stochastic Performance Analysis Model</th>
<th>149</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 8.1</td>
<td>Probability distribution of production rules</td>
<td>156</td>
</tr>
<tr>
<td>Figure 8.2</td>
<td>The context-free grammar for the X.25 layer 2 receiver entity</td>
<td>159</td>
</tr>
<tr>
<td>Figure 8.3</td>
<td>Symbol processing timing diagram</td>
<td>160</td>
</tr>
</tbody>
</table>

| Chapter 9 | Conclusions and Directions for Future Work | 165 |

- xiv -
### List of Symbols

#### Chapter 3

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFG = (V_T, V_N, S, P)</td>
<td>a context-free grammar</td>
</tr>
<tr>
<td>AG = (CFG, A, D, AT, VA, SA)</td>
<td>an attributed grammar</td>
</tr>
<tr>
<td>PG = (AG, TA, PC)</td>
<td>a protocol grammar</td>
</tr>
<tr>
<td>L</td>
<td>language defined by a grammar / implemented by an automaton</td>
</tr>
<tr>
<td>p, P</td>
<td>production rule, set of all production rules</td>
</tr>
<tr>
<td>AS</td>
<td>attribute scope</td>
</tr>
<tr>
<td>AM</td>
<td>attribute mode</td>
</tr>
<tr>
<td>W</td>
<td>a finite set of denotable values for a protocol grammar</td>
</tr>
<tr>
<td>BE_n</td>
<td>a finite set of boolean n-ary functions</td>
</tr>
<tr>
<td>AE_n</td>
<td>a finite set of arithmetic n-ary functions</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>a finite set of attribute names for a protocol grammar</td>
</tr>
<tr>
<td>D</td>
<td>a finite set of data types for a protocol grammar</td>
</tr>
<tr>
<td></td>
<td>length of a list x</td>
</tr>
<tr>
<td>x_i</td>
<td>i-th element of list x</td>
</tr>
<tr>
<td>I</td>
<td>projection operator (onto alphabet)</td>
</tr>
<tr>
<td>⊗</td>
<td>concatenation operator (lists)</td>
</tr>
<tr>
<td>ξ^R</td>
<td>reversed list ξ</td>
</tr>
<tr>
<td>⊥</td>
<td>'unallocated' value</td>
</tr>
<tr>
<td>T</td>
<td>'undefined' value</td>
</tr>
<tr>
<td>E</td>
<td>set of attribute assignment expressions over W</td>
</tr>
<tr>
<td>Σ</td>
<td>unprocessed symbol with attributes and expressions</td>
</tr>
<tr>
<td>OS_j, IS_j</td>
<td>j-th finite output/input symbol alphabet</td>
</tr>
<tr>
<td>OS, IS</td>
<td>the total output/input symbol alphabet (all channels combined)</td>
</tr>
<tr>
<td>O_j, I_j</td>
<td>j-th output/input channel alphabet (symbols with attribute values)</td>
</tr>
<tr>
<td>O, I</td>
<td>the total output/input channel alphabet (all channels combined)</td>
</tr>
<tr>
<td>A_g, A_l</td>
<td>set of all global/local attributes in PG</td>
</tr>
<tr>
<td>ENV</td>
<td>set of attribute environment functions for a protocol grammar</td>
</tr>
<tr>
<td>θ</td>
<td>function that extracts all local attributes from a sequence of unprocessed symbols</td>
</tr>
<tr>
<td>ρ[a / w]</td>
<td>value assignment of value w to attribute a in environment ρ</td>
</tr>
<tr>
<td>EV</td>
<td>semantic evaluation function for assignment expressions</td>
</tr>
</tbody>
</table>
List of Symbols (continued)

CV  semantic evaluation function for condition expressions
Z  rule reshaping function
PR set of reshaped production rules
<b, n, η> reshaped rule for a PG: b = condition, n = LHS, η = RHS
<t, α, ε> unprocessed input or output symbol
<a, α, ε> unprocessed nonterminal
χ, X unprocessed endmarker, its type
α, α₁, α₂, ... lists (mostly of attributes)
ε, ε₁, ε₂, ... lists of attribute evaluation expressions
c, c₁ condition (boolean) functions
r_k the kth reshaped production rule
< >_GI protocol grammar input configuration
< >_GO protocol grammar output configuration
φ ∈ (I ∪ O)* a sequence of accepted inputs and generated outputs
β ∈ (Σ ∪ X)* a sequence of unprocessed symbols
ω = φβ a leftmost sentential form
δ = <ω, ρ> a leftmost derivation configuration
Δ set of all leftmost derivation configurations
Y ⊆ Δ × Δ a leftmost derivation step
Δ̃, Δ̃̃ infix notation for Y, its reflexive transitive closure

Chapter 4

M = (Q, IAS, OAS, U, H, Q₁, Q₂, A_m, W_m, C) a protocol automaton
Q a finite set of states
IAS a set of k finite input tape symbol label sets {IA₀, ..., IA_k-1}
OAS a set of n finite output tape symbol label sets {OA₀, ..., OA_n-1}
U a finite stack alphabet
H program, a finite set of instructions (defined later)
Q₁, Q₂ the set of initial and final states
A_m a finite set of so called environment variables (names)
W_m a finite set of denotable values for a protocol automaton
C a function mapping any label from any IA_i or OA_i to an integer
B set of boolean values {true, false}
List of Symbols (continued)

IA, OA the total input and output label sets (all tapes combined)
Ξ, Ψ total input and output tape alphabet (with attributes)
ENVₘ set of environment functions for a protocol automaton
< >ₘι protocol automaton input configuration
< >ₘo protocol automaton output configuration
γ, Γ machine configuration, set of all machine configurations
q ∈ Q a state of the FSM of a protocol automaton
qₖ state where nonterminal is popped to be expanded with rₖ
q₋ₖ, q₀ state in which input/output is processed
qₐₙₙ state in which LHS of rₖ is pushed on the stack
ξ ∈ Ξ* a sequence representing an accepted machine input configuration
ψ ∈ Ψ* a sequence representing a generated machine output configuration
σ ∈ U* contents of the stack
∈ M a move of a protocol automaton
→ M, → MC a sequence of moves of a protocol automaton
← R bijective relation between leftmost derivation configurations and machine configurations

Chapter 5

L, map function mapping attributes to memory locations
ind fixed function mapping memory locations to other locations
m, mem function mapping memory locations to denotable values
Locn a memory location, either absolute or base with index

Chapter 6

MS = (K, P, R, C, B) a memory manager state
K memory page size
P finite set of pages (numbers)
R finite set of block references
C function that returns the user count for each page
B function mapping references to memory blocks
List of Symbols (continued)

- $m, M$: memory block, memory block type
- $Q$: list of pages
- $V$: value function (maps memory locations to values stored)
- $L$: maps block reference with offset into physical memory location

**PS**: $(M, S, W, F)$ a packet manager state

- $W$: finite set of packet reference numbers
- $F$: function mapping packet reference numbers to actual packets stored in memory
- $D$: packet stored in memory
- $u, U$: unit (part of packet), unit type

### Chapter 8

- $\|p\|$: the number of symbols on the right hand side of rule $p$
- $\lambda$: probability distribution function over the production rules
- $\varepsilon$: empty string (string with length 0)
- $S_{ij}$: $j$th symbol of the $i$th rule
- $\tau_{ij}$: time required to process symbol $S_{ij}$
- $T_N$: time to completely reduce a nonterminal $N$ to $\varepsilon$ in a grammar $G$
- $E_G[T_N]$: mean value of random variable $T_N$
- $T^\text{search}_N$: average time required to find a rule for expansion of $N$.
- $T^\text{exp}_i$: time required to expand a nonterminal using rule $i$ (constant).
- $\#t_N$: the number of terminals $t$ encountered in a complete reduction of nonterminal $N$
- $E_G[\#t_N]$: mean value of random variable $\#t_N$

**Note:** Any indexed symbols or symbols followed by one or more single quotes are of the same type as the plain symbol.
The advances in computer technology have led to an enormous increase in the use of computers. Since computers are basically data processing devices, it was inevitable that digital communications started to play an essential role. Today, most computers are connected to a network over which they can exchange data. The method of communication, the rules involved in the regulation of data transport and the type of network can have a major impact on the performance of a computer system.

1.1 Data communication related problems

1.1.1 Transmission errors

The communication or exchange of data between computers requires transmission and reception of signals. These signals traverse a medium between the transmitter and the receiver, arriving at the destination in a more or less distorted form. The medium is usually a wire, but could also be fiber (less sensitive to interference), satellite links or radio waves (more sensitive) or any other medium for signal propagation. Signals are distorted when traversing a medium for various reasons:

- Energy loss due to transmission line effects
- Interference from external sources
- Frequency dependent delays
- Other non linear properties of the medium

Every one of these effects can make data transmission unreliable. To ensure a reliable data transmission, even on an unreliable medium, several methods are available such as data encoding using error detection and/or correction codes. Uncorrectable errors require retransmission of the data, otherwise data is lost.
1.1.2 Medium access control

Some media can only transfer signals from a single source at a time. If more than one transmitter is connected to a network using such a medium, an arbiter mechanism for regulating transmitter access is required. This mechanism can be implemented on a single machine which controls all other transmitters, or it can be distributed over all or some of the transmitters [Tasaka86].

1.1.3 Communication terminology

Computers can only communicate if they use the same language, i.e. if commands, responses and data are encoded using an unambiguous predefined or negotiated method. Only then can the receiver interpret and process received bit patterns exactly as they were meant to be. The coding scheme used for the communications must be identical for all pairs of stations that need to communicate [Barnett88, Bonatti87].

1.2 Communication protocols

1.2.1 Informal definition

The precise methods and codes involved in solving the problems mentioned in the previous section and many more aspects regarding the communication procedures are specified in standards called protocols. There are many different and mutually incompatible protocols available for the design of communication systems, and new ones are still being developed. Informally, a protocol is a set of rules to which all relevant input and output actions of a system must comply. Two systems built according to the same (correct) protocol can communicate because they use the same codes and rules to control data transfer.

1.2.2 Hardware and software implementations

Practical protocols are usually very complex and hard to specify in natural language. For example, the latest revision of the widely used X.25 protocol is more than 150 pages of text, tables and codes [Deasington85, FIPS87]. This also makes these protocols hard to understand. Not surprisingly, nearly all implementations of such protocols are made in software by creating algorithms whose input and output actions observe the protocol and executing those algorithms on computers. Advantages of software implementations over hardware are flexibility (easier to make changes), lower production costs, easier implementation, etc. A major disadvantage is the rela-
tively low communication speed that can be obtained when compared to hardware implementations. This is caused by the lack of true parallelism in software (unless hardware supports it) and the overhead of machine level instruction coding. An algorithm would generally need multiple instructions (machine cycles) to get a result that could be obtained in one cycle with dedicated hardware.

1.2.3 Protocol classes

Protocols are usually divided in two distinct classes: character (byte, octet)-oriented and bit-oriented protocols (see par. 1.2.3.1 and 1.2.3.2). The protocols from the two classes have very different characteristics and therefore applicability. The character-oriented protocols are mainly used for (short distance) communication between two devices and not on networks. Bit-oriented protocols can be used for almost any purpose and must be used on networks and over long distances. The next two sections summarize the most important features and (dis-) advantages of both classes.

1.2.3.1 Character-oriented protocols

In character-oriented protocols, the smallest transmittable entity is a character (byte). Every character is a stand alone entity that must be interpreted as such by the receiver. The value of a character determines its meaning. Special values are used for command exchange between transmitter and receiver (control). The main characteristics are

- one alphabet is used for both data and control information.
- control and data can be mixed randomly or according to some predefined scheme and are distinguishable only because of their values or this scheme. In some cases (e.g. if a character is lost) data could easily be mistaken for control information.

Character-oriented protocols do have a number of disadvantages:

- The control codes and sequences for control and link management are usually not protected by any error detection mechanism.
- Each transmitted (data or control) unit serves only one function (e.g. data block marker, acknowledge, enquiry).
- The freedom in interpretation for the defined control codes has led to big differences in the applications of these codes in various systems. The standards have virtually vanished.
1.2.3.2 Bit-oriented protocols

In bit-oriented protocols, the smallest transmittable entity is called a packet or a frame. A packet is a sequence of fields, which in turn are sequences of bits. The length of each field is usually a multiple of 8 bits (octet), but this depends on the protocol definition. The meaning of a bit depends on both its value and its location within the packet.

- Control information is always located at the same position in transmitted frames (it can never be mistaken for data or vice versa, if the frame arrives at the destination without errors).
- Controls are bit patterns that are independent of any chosen data alphabet, so data transmissions are transparent.

The bit-oriented protocols were developed because of the major drawbacks of the character-oriented protocols. They are supposed not to possess any of the negative properties of those protocols. The following demands were taken into account:

- Full data transparency (any bit pattern can be transmitted without danger for control and data intermixing).
- Adaptable to most applications in an easy and consistent way.
- High efficiency (i.e. low overhead/multiple function transmission units)
- High reliability.

1.2.4 Protocol standardization

For both classes of protocols there are a number of worldwide standards. These standards are defined by international committees and revisions are submitted every few years. Because of their disadvantages, the character-oriented protocols are becoming outdated for new high speed applications. Most new protocols are derived from the standards proposals for bit-oriented protocols, made by the International Organization for Standardization (ISO) [Sunshine89]. This currently accepted proposal describes a layered architecture for protocols called Open Systems Interconnection (OSI). It is obvious that research on protocol engines should be directed towards the implementation of bit-oriented protocols. The main reason for such an engine would be the vastly increased speed of data transfer, and character-oriented protocols are just not suitable for high speed communication. The remainder of this thesis will therefore concentrate on bit-oriented protocols and their implementations.
1.2.5 Concepts of the OSI reference model

1.2.5.1 Layered architecture

In the OSI reference model, a protocol is represented as a set of 7 hierarchically ordered layers. Each layer provides additional functionality to higher level layers (except layer 7) and hides unwanted properties of lower level layers (except layer 1). The lowest layer controls the medium and completely shields all other layers from the medium and its specific access control. The highest layer is the application layer, where data generating and accepting applications (software) are located. The set of 7 layers provides a completely shielded communication system to the applications, where all communications can be handled in the same way, regardless of the physical location of the peer application or its type of system (heterogeneous networks). The advantage of dividing the protocol in layers is that each layer only performs a specific subfunction and is therefore easier to describe and implement. The layer model of OSI and the names used for the layers are shown in figure 1.1.

![OSI Reference Model Diagram](image)

Figure 1.1 The OSI communications model.

1.2.5.2 Naming conventions

The active elements within a layer are called entities. Entities in the same layer are called peer entities, regardless of whether they are active in physically separate or different types of communicating systems. Peer entities cooperate to provide a certain service, for which they must interact according to some rules. This set of rules is called a protocol. A layer offers services to the next higher layer and uses services offered by the layer directly below it. Communication between 2 adjacent layers
(requesting services, returning responses) takes place over service access points (SAPs). To distinguish between entities and services in or between different layers, the layer number will be put in front of the entity or service: (N) entities are the entities in layer N, (N) SAPs are the service access points that interface the (N) layer to the (N+1) layer. Peer entities are connected by (N) connections as special part of the (N) services offered by the (N) layer. These (N) connections allow the (N) entities to exchange messages for their cooperation. (N) Connections may be either point to point (e.g. SAP to SAP) or multi-endpoint. The endpoints of (N) connections at their respective (N) SAPs are called (N) connection end points or simply (N) CEP. See figures 1.2 and 1.3 for a graphical representation.
1.2.5.3 Service primitives

At the boundary between layers N and N+1, the entities in layer N provide services to layer N+1. To perform these services, the (N) entities will in turn use services provided by layer N-1, and so on. For the description of (N) services, the lower layers up to and including the medium can be regarded as a black box. Only the definition of the services available to layer N is required to be able to use this black box. The boundary between layers is defined in both directions: an (N) entity can also inform or request a service from an (N+1) entity. These interactions are based on the concept of service primitives. With respect to services entities can be divided into service users and service providers. Every entity can act as a user, and together with all lower layers it can act as a provider. Each kind can issue certain types of primitives. There are 4 types of service primitives:

1) Request primitives, issued by a service user to a service provider to request invocation of a service
2) Indication primitives, issued by a service provider to request a service from another service provider or to indicate the acceptance of a request from a service user on a peer SAP.
3) Response primitives, issued by a service user to indicate that a service requested by an indication is completed.
4) Confirm primitives, issued by service providers in order to indicate to a service user that its requested service is completed.

1.2.5.4 Service Data Units and Protocol Data Units

The data units that are used to transport information between adjacent layers are called Service Data Units (SDU), and those passed between peer layers are called Protocol Data Units (PDU).

![Diagram](image-url) #1.4 Naming of data units in layers.
Figure 1.4 shows the relation between these units in the simplest case, where no additional length conversion operations take place. Layer N+1 transmits a PDU to a peer layer in another station by sending it as a SDU to layer N. Protocol Control Information (PCI) is prepended for the peer layer N and the result is transmitted to the peer layer N as a PDU. Figure 1.5 shows the conceptual difference between PDUs and SDUs.

Packet conversion operations in entities can split or merge incoming SDUs before adding PCI, and split or merge PDUs before actually sending them to the next layer. In these cases there is also a real difference between SDUs and PDUs. Operations like these can become necessary if a network does not support packets of arbitrary length, i.e. if packets would otherwise be too short or too long.

![Figure 1.5](image)

**Figure 1.5** *PDUs are sent between peer entities and SDUs between adjacent layers.*

### 1.3 Protocol Implementations

#### 1.3.1 Hardware versus software implementations

Protocol implementations can be classified as hardware or software implementations. The difference is not always clear and an exact definition cannot be given. For example, when programmable components are used (PROMs, PLAs, etc.), would it be considered hardware or software? What about the case of a dedicated microprocessor running a specially written microprogram?

In this thesis a software implementation is a system where no dedicated hardware is used and where the protocol is completely rewritten in algorithmic form to obtain machine executable code. A hardware implementation extends the machine with dedicated hardware that performs most or all of the actions of the protocol, even
when some of its parts are programmable. The two examples mentioned before would therefore be classified as hardware implementations.

Communication protocols are often implemented in software for a very large part. Layer 1 (the physical layer) has to be implemented in hardware, since it must interact with the physical medium. Only for some protocols which are used extensively throughout the world, such as ethernet, and for some high performance protocols that were especially designed to allow efficient hardware implementation, chip sets have become available for some or all layers. The reasons why in general software is used for protocol implementations are:

- The high complexity of the protocols, which makes it very hard to design any implementations at all and specifically in hardware.
- The flexibility of software implementations. It is much easier to make changes to a software implementation and it is also easier to handle special cases.
- The tools available for designing, testing and debugging complex software systems are much better developed than those for hardware.
- The costs of software implementations is generally lower. Once an implementation is made, it can be duplicated as many times as desired at almost no additional cost.

1.3.2 The software bottleneck

Software implementations have one big disadvantage: achievable communication speeds are much lower than that of dedicated hardware implementations. Until recently this was not a problem, since host computer systems were fast enough to run protocol software that could utilize the entire available bandwidth of networks. Advances in telecommunication technology have led to new types of networks (fiber optics) whose capacity is much greater, and it is expected that this capacity will grow even further in the near future. However, the processing power of computers has not nearly increased as much, and software implementations will no longer be capable to fully utilize the available bandwidth. Instead, the communication speed will be determined almost completely by the speed at which software is executed on the host system [Chesson87]. The complexity of the protocols introduces so much overhead that the software becomes a bottleneck in performance. It will therefore become necessary to implement protocols (for a larger part) in hardware.

The advantages of software over hardware implementations must somehow be overcome, which basically means that a tool must be developed that enables the development of hardware implementations for complex protocols in a comprehensible way.
Complexity can be reduced by using the layer architecture of most protocol stacks. Every layer of the protocol stack can be implemented separately and independently and it might also be possible to further subdivide layers. To maintain flexibility, at least some of the hardware (the parts that control the execution) will have to be programmable. To reduce costs, the same architecture should be reusable for many different protocols, which allows cost efficient implementation in the form of programmable integrated circuits.

1.3.3 Other research, work and products for protocol implementations

The ideas that automatic implementations are desirable and that hardware implementations can yield higher throughputs (>> 100 Mbit/s) is not new. Many projects have been started by research institutes, and some semiconductor manufacturers have developed special chip sets to implement certain parts of protocols in hardware.

AMD has created a family of 5 devices for FDDI based networks, the so called Supernet™ Family. These 5 devices can be used to implement the ANSI X3T9.5 token passing protocol at 100 Mbit/s transfer rate. Furthermore, special purpose chips have been developed for certain subtasks of some (high level) protocols, such as encryption and data (de-)compression. They also have ISDN controllers, but only for low speeds (a few Mbit/s).

In [Chesson87], some interesting work is presented on the implementation of custom protocol engines for speeds up to 100 Mbit/s, supporting a variety of physical layers (FDDI, ethernet, etc.). It uses a custom protocol (i.e. packet formats) to enable efficient implementation in VLSI, and can therefore not be used for any (general) protocol.

In [Partridge90], it is argued that transfer rates of up to 1 Gbit/s can be obtained with current technology (a set of RISC processors per implementation) for all kinds of protocols (X.25, TCP/IP, etc.). A number of related problems are mentioned, regarding buffering, delays, and flow control, but no actual architecture or method for the creation of one is given.

[Martini89] introduces a transputer based architecture, where protocol layers are implemented on separate transputers, connected to a shared data memory. Although transputers are fast, they will always be slower than dedicated hardware made in the same technology.
A specialized switch architecture for ATM has been developed and presented in [Fried89]. It employs a very fast small microprocessor with only a few instructions to program its operation.

In [Popescu-Zeletin88], a modular gateway architecture is proposed to achieve high speeds in future ISDN-based networks. Special processors (transputer arrays or microcontrollers) are mentioned as possible implementation methods, but no indications of performance or any analysis is given.

In [Hansson86], a tool is presented for the automatic implementation of protocols in a pascal program.

A universal programmable hardware architecture and a method for mapping many protocols to it is given in [Krishnakumar87]. To implement a protocol, it must first be described as a set of finite state machines, which are then automatically mapped to the architecture. This can be done fast, easily and cheap. However, the system is quite simple and apparently cannot reach high speeds (up to a few Mbit/s).

In [Schindler79a] and [Schindler79b], a machine organization is proposed that can be used as a basis for protocol implementation in general. Some of these ideas will be used in this thesis.

Other projects for LAN protocol implementations are presented in [Sharp87], [Jensen88] and [Rupprecht88]. These are directed towards specific protocols or functionality.

In [ZitterBart90], an architecture for general protocol implementation is given, based on a set of parallel operating transputers. Special pipeline and processor array architectures are considered. An 8 transputer system managed to switch approximately 2500 packets per second, which is considered slow by the author.

From this short and incomplete survey, one important conclusion can be drawn: most of current research and development work on implementation of protocols (automatic or not) is directed towards (a) very specific protocol(s) which are specially designed for 'easy' implementation, towards low data rate protocols (ISDN ≤ 192 kbit/s, ethernet, etc.), or towards very specific functionality (bridges, protocol converters). The research which is done on high speed general protocol implementations either remains abstract (no architectures or solutions given) or it uses microprocessors on which the protocols are implemented entirely in software.
This thesis presents research work that is directed towards bridging this apparent gap: full automatic implementation in hardware of any general data communication protocol from a formal protocol description, such that extremely high data rates can be achieved.

1.4 Protocol engine concepts

A typical configuration for a data communication environment is shown in figure 1.6. This clearly shows the position of the protocol implementation in the system as Data Termination Equipment (DTE). At one side it connects to the data processing equipment (DPE) which is usually a host system. The DPE is assumed to be fast enough to provide a maximum load for the protocol engine. At the other side the DTE is connected to a DCE that terminates the network. The DCE interfaces to a medium and it must therefore implement a corresponding medium access protocol. The DCE shields the protocol system from the medium and is considered part of the first (physical) layer of the protocol.

![Diagram of communication system]

Figure 1.6 The position of the protocol engine in a communication system.

For the protocol engine research the DCE is not of interest. It will be regarded as part of the network, since it does not change or interpret any messages but merely passes them to the medium. The network is a given entity that cannot be changed. The DCE provides an interface where all network dependencies are hidden. It will be assumed that the DCE accepts packets generated by the protocol system directly. Sometimes the DCE can fill in network addresses (for transmissions), but here this function will be assigned to the protocol system.
For most bit-oriented protocols, the implementation can be partitioned in a high speed front end and a variable or programmable part, as shown in figure 1.7. The front end is a fixed architecture implementing the bit level functionality of the physical and datalink layers. This includes operations such as parallel/serial data conversion, CRC checksum generation and checking, bit stuffing and destuffing, and synchronization flag insertion and detection. The front end is basically an interface to the DCE that relieves the protocol engine from the standard bit level encoding operations to prepare packets for transmission on a medium. A front end architecture that is usable for protocols with a HDLC datalink layer is given in figure 1.8.

![Figure 1.7 Partitioning of the protocol implementation.](image)

The more interesting part is the Protocol Processing System, where the actual processing and generation of packets takes place. This part of the implementation will be the target for automatic implementation by the protocol engine compiler described in this thesis.

![Figure 1.8 A front end system for protocols using HDLC.](image)
1.5 Research aim

The design of a hardware implementation of a communication protocol, a so-called protocol engine, is a very complex task. It requires amongst others knowledge on general protocol functionality, on methods of implementing this functionality and on description or specification languages. It would be a great step forward if implementations could be generated automatically from a description of the protocol. The development of a system that can do this will be the main research goal.

The architecture of the protocol engine could either be fixed and programmable (in which case a specific protocol is mapped to this architecture) or variable (in which case custom hardware is generated for a given protocol). The fixed architecture has the advantage that later changes can be incorporated after the engine has already been built without extensive hardware changes. It will even become possible to make a standard protocol engine, that can be programmed to execute almost any protocol. The variable architecture approach would probably generate totally different (and probably faster) hardware for each protocol, and it is also a lot more complex. Making changes to such an implementation afterwards is more difficult. To get the best of both methods, an intermediate solution is desired: a variable architecture which consists of elements from a fixed set of basic modules, each of which can be programmed or adapted in certain ways to meet the requirements of a particular protocol.

1.5.1 A protocol engine design system

The automatic creation of a hardware implementation for communication protocols leads to the requirement of a formal description language. Any protocol to be implemented should first be described in this language. A software tool can then analyse this description and create an implementation. The language must be formal and powerful enough to express any required protocol functionality. It should not be too abstract to make implementations difficult to generate.

Hardware implementations are created by mapping formal descriptions to modules from a hardware library and connecting these modules in a way derived from the descriptions. Each library module is a parametrizable architecture, some of which are also programmable. The parameters can be set to adapt the module for certain requirements. Programmability means that the functionality of the module can be changed by providing a list of instructions to be executed by it.
The creation of a protocol engine consists of two phases. During the first phase the formal descriptions are read and analysed, and from these the parameters and programs for the modules are determined. During the second phase the actual modules are retrieved from the library, parameters are set, program code is loaded and all modules are connected in the correct way.

To reduce complexity of the descriptions, the protocols can be divided into layers and even further into entities within layers. A protocol is then described as a set of interconnected communicating entities, each of which is described in the formal language and implemented separately.

1.5.2 Research areas within the project

The project can be divided in 5 research areas:

- A library of parametrizable and/or programmable architecture elements, that can be used to construct a protocol engine for OSI-type protocols.
- A formal language suitable for the description of protocols in an abstract way, but such that translation to hardware can be performed automatically.
- A compiler for the description language, which generates parameters and all code for programmable library elements from a description.
- An engine constructor that creates a complete architecture using the outputs of the compiler.
- Integration of all parts into one complete protocol engine design system.

It is expected that the step towards hardware implementations will fill the gap that is now being created between high speed networks and host machines due to the advances in network technology. It is the opinion of the author that protocol engines are (in the near future) the only way to fully utilize bandwidth on coming Gigabit networks.
Chapter 2

Formal Protocol Specification Languages

Protocols describe rules and mechanisms for the interactions of systems. The language that is used for this purpose has a large impact on specification, verification, and implementation of protocols. For some time natural languages were used. These informal methods have led to a large number of errors in many protocols. With the growing complexity of modern data communication protocols, informal specification techniques are no longer considered sufficient and a number of formal techniques have been invented. This chapter presents generally accepted requirements for protocols and protocol specification languages. It also introduces formal grammars as a formal technique for the description of protocol implementations and it indicates how these grammars can be used to achieve the research goal.

2.1 Protocol development

Informal methods for protocol specification are no longer suitable, since such specifications are often ambiguous and incomplete. This does not mean that informal descriptions are useless, however. Descriptions in plain natural languages are very helpful in obtaining a general understanding of the operation of a protocol. The problem is that the ambiguity introduced by these informal languages results in incompatible implementations of the same protocol, because different implementors interpreted the specification in different ways. Other errors, such as incompleteness (where a specification neglects to treat all possible system inputs) are hard to detect by hand and nearly impossible by automated methods if the specification itself is in an informal language. Tools for automatic analysis, verification and implementation of protocols are only possible using formal specifications. A formal language is a language for which a one-to-one mapping from the syntactical to the semantical domain has been defined. From this mapping, the semantics of every possible valid
expression of the language can be determined. Two or more implementations from the same formal specification should always be compatible.

2.1.1 Correctness of protocols

An implementation is generally correct if it meets its specification. This kind of correctness can be established by mathematical analysis if implementation and specification are both formal systems. In other cases, testing is a method for finding certain implementation errors, but it is not guaranteed to find all errors. Although verification methods can show that an implementation is correct with respect to its specification, they do not tell whether or not the specification itself is correct.

A set of requirements has been defined that can be considered an implicit part of every protocol service specification. These requirements have to be fulfilled for a correct specification. Most verification techniques are directed towards proving them for a given specification. The requirements are:

- **Absence of danger of deadlock.** This means that there should be no reachable protocol state from which there is no exit (except for final states).
- **Progress.** There may be no useless cycles (where no progress is made) in the specification. It must also guarantee that both communication partners make progress.
- **Completeness.** All possible inputs must be handled somewhere.
- **Stability.** The protocol should always return to some basic state within a finite amount of time, independent of the behaviour of the environment. This implies tolerance for (and handling of) errors made by the communication partner or the environment.

Any implementation must satisfy the following two global demands:

- **Safety.** This means that all actions that take place (if any) are according to the protocol specification.
- **Liveness.** Any requested service must be completed or terminated according to the specification within a certain time limit.

A protocol is correct if it satisfies the above requirements.
2.1.2 Protocol layers

Most formal techniques are based on (or directed towards) protocols that consist of a hierarchically ordered set of layers. This communication architecture allows for a separate specification of each layer. Every layer offers a certain set of services to the adjacent higher layer and uses the services provided by its adjacent lower layer to achieve them. The boundary between two adjacent layers is called an interface. For the specification of a single layer, the lower layers can be regarded as a black box which can provide certain services; the mechanisms by which these are achieved are unknown and irrelevant. To a user (a higher layer), only the messages that cross the interface and their causes/effects are of importance.

The layer specification is a specification of all services provided by a layer. This usually involves the introduction of so called layer entities (functional blocks within a layer) which interact with each other and with entities in other layers by exchanging commands and responses, which are generally known as service primitives. A layer specification is an abstract definition (containing only essential information) that consists of 2 parts:

- **Service specification**, used to define the service primitives. All commands, responses and their effects are described in terms of services provided by lower layers and functionality offered by the entities. Note that this does not define how a certain service is achieved, but only its effects. It also does not specify how to invoke a service (conveyance of primitives).
- **Interface specification**, used to define the formats of service primitives that can be used to convey commands, responses and other messages over the interfaces of the layer. It does not specify how these primitives should be exchanged (implementation detail). Only the data formats are defined.

A formal protocol specification language should have constructs that allow the definition of these two parts in addition to timing elements. The three basic elements for any specification are therefore (lit. [Sunshine89]):

- **Syntax.** This element defines the types of commands, responses and other messages, as well as their formats.
- **Semantics.** Defines the relation between the commands and responses, causes and effects (i.e. the complete I/O behaviour).
- **Timing.** Defines duration of actions, delays, time-outs, etc. This imposes an additional temporal ordering and constraint on behaviour defined by the semantics.
2.1.3 Design trajectory

The development of a new protocol starts with an idea, which is first written down in informal textual form. Much effort then goes into obtaining a precise definition of the protocol in a textual form, called the reference model. The next step is the creation of a formal specification of the protocol. This specification has to be verified with the reference model. Both the reference model and the formal specification may require updating. The whole process iterates until a correct formal specification is found that satisfies the reference model.

If the goal is to implement the protocol, then the specification is gradually transformed into increasingly detailed descriptions (by replacing abstract definitions by less abstract ones and making implementation choices), until an executable or otherwise implementable description is reached. At every step, the new description must be verified against the formal specification. This will ultimately result in an implementation that satisfies the original formal model, and which is therefore correct.

The steps of the design trajectory are by no means trivial. If any of these steps could be omitted, automated or in any way made easier, it would be a welcome addition to protocol development techniques. The method presented in this thesis is aimed at completely automatic implementation of protocols in hardware from a formal specification.

2.2 Properties of protocol specification languages

Formal description languages must have a number of special properties to be suitable for protocols. The demands for specification are different from those for implementation. The remainder of this section lists the most important properties for specification languages, taken from [Gelli87], and whether they are also required if the goal is to make an implementation instead of a specification.

To be useful for the specification of protocols, a formal specification language must

- always have a formally defined syntax and interpretation model (semantics). For implementation purposes, there should be a mapping from syntax to this model.
- support concurrency, communication (both synchronous and asynchronous, as well as abstraction from the type of communication), synchronization and interleaving. For implementation purposes, concurrency is desirable but not
required (makes implementation more difficult).

- include timing aspects and constraints, such as action/reaction interval specification, relative timing and wait (delay) constructs.
- allow nondeterminism with and without fairness. This is not required for implementation purposes.
- have integrated data specification, which is strongly typed. It should use abstract data types (ADTs) and allow incremental definitions. For implementation, it is easier not to use ADTs, because they are difficult to implement.
- support multiple levels of detail, allow refinement or subdivision of constructs with equivalence checking.

Furthermore, it is desirable (but not absolutely required) for a protocol specification language to:

- have associated tools to support the operations: syntax/semantics checking, correctness verification, simulation, symbolic evaluation, implementation, debugging, test and report generation and performance analysis.
- allow the checking of consistency of the protocol (both internal and external), its completeness, the absence of deadlock and the equivalence of two specifications.
- be modular, extendible and support multiple paradigms, e.g. procedures, macros, abstract data types (ADTs), finite state machines, etc.
- be machine independent, i.e. not contain any implementation details.
- allow specification of the control flow (execution) and separate error/exception handling.
- be easy to use and understand by humans: allow redundancy and comments, use implicit declarations and have a simple syntax.

### 2.3 Formal Grammars

A number of formal description techniques for (general) systems have been created over the past decades. The best known are:

- Finite state automata
- Petri nets
- Formal grammars
- Process oriented languages
- Programming languages
- Temporal logic
Various extensions on the basic models of these formal description techniques have been invented as well to allow easier or better expression of protocol functionality. The most widely used methods for specification are finite state automata (FSA), petri nets (a graphical state oriented method) and the process oriented techniques. Programming languages are used mainly for implementation and temporal logic is used primarily in conjunction with other techniques, such as FSA, to express the timing aspects of systems. Formal grammars have been the topic of research on formal protocol specification techniques a few times in the past. Despite the positive results (to the author's opinion), an accepted method for grammar based protocol specification has not yet resulted from that research.

The use of grammars does have some advantages over finite state oriented methods. Many protocols have complex execution control loops, which are easier described by recursion than by iteration. In designing and specifying protocols it is often desirable to be able to abstract from certain fixed values, maximum recursion or iteration depths, etc. This means that the ability to express unbounded recursion is a welcome addition to any protocol specification language. Grammars (except regular grammars) offer this possibility, whereas FSA based methods do not. Furthermore, when attributes are associated with the symbols of the grammar to store context information (semantics, see [Knuth68]) the attribute space, which contributes to the total system state, varies in length as attributes are created and deleted with the corresponding symbols. When recursion is unbounded the attribute space can also grow without upper bound. Such a state cannot be embedded in the control vector of a finite state automaton which must have a known finite length.

Process oriented techniques also offer unboundedness, but it is much harder to translate such a specification to hardware implementations. Grammars, which are action oriented, are considered to have a lower abstraction level than process oriented techniques, but a higher one than FSA methods. As a consequence, grammar based specifications are harder to translate to hardware implementations than FSA methods, but easier than process oriented techniques.

The use of formal generative grammars to model protocols is based on the following point of view. The input and output actions of the protocol system can be regarded as words of a formal language. The valid sequences of these actions are the sentences of the language. Such a language can be defined by a grammar, where the I/O actions are represented by the terminal symbols and the definition of valid sequences is given by the production rules (lit. [Denning78]). The grammar definition abstracts from all internal actions, data and functionality and concentrates com-
pletely on externally observable behaviour. It directly specifies what I/O actions (sequences) are allowed, but it does not tell how this behaviour should be achieved. This corresponds very closely to the concepts explained earlier regarding layer definitions.

A system that can recognize the language defined by a grammar is often called a parser. A system that emits sentences from such a language is called a generator. Techniques for building parsers are well known and can be automated. The differences between normal parsers and a protocol parser/generator are:

1) Sentences are not given in advance but are constructed interactively between two communication partners.
2) Time delays between successive inputs are also important for protocols.
3) A protocol is both a parser and a generator since it accepts and generates messages belonging to the language specified by the protocol grammar.

Although desirable, this work is not directed towards optimal implementation using some hardly readable description language, but towards finding a reasonably good and completely automatically generated implementation from an easily (human) readable and usable language (a compromise).

A grammar is to the author's opinion a very intuitive way of describing protocols, since it directly defines allowed sequences of (I/O) actions (generative description), instead of some algorithm that implements the protocol (FSA, program, etc.). The idea is to have nonterminals of the grammar (see chapter 3) derive logically related sets of events (phases of a protocol). This constitutes a top-down description and design method for protocols (see also [Anderson85a/b, Harangozó77/78] and [Haas85/86]).

2.3.1 Automatic implementation

The advantages of (partly) automatic implementation are obvious: there are no (or less) errors and it also becomes much easier to make changes since it only requires the recompilation of a description. Furthermore, the same description can be compiled for many different machines (architectures) which will then all behave in a compatible way. Automatic generation is much faster than hand coding or designing, resulting in much lower development costs. In case of grammars, a parser needs to be constructed only once for every machine, and can then be used for many different protocols. New protocols are implemented by simply specifying new parse
tables, etc. These are independent of the machine on which the parser is running. Changing a protocol, even when it is already in use becomes much easier.

The theory of languages and automata shows that grammars can be divided into a number of classes, and that for each class there exists a type of (abstract) automaton that can implement any grammar from that class (lit. [Denning78]). For systems with a finite number of states, the grammars can be constructed such that these automata are finite and deterministic as well. This means that the problem of finding an implementation in hardware is already solved and that the solution has been mathematically proven correct. The well known compiler construction techniques are all based on formal grammars (lit. [Fisher88] and [Aho86]), which means that a software implementation (parser) can also be generated from a grammar. This allows easy and fast implementation in both software and hardware from the same model. Making a change in the protocol requires changing the grammar and then recompiling the implementation (software and/or hardware), resulting in very small turn around time and fast availability of test applications.

2.3.2 Verification and testing with grammars

Apart from automatic implementation, the formalism of grammars may allow other tools as well. Verification by symbolic execution is a possibility, because of the simplicity of grammar descriptions. Although the extensions will make automatic symbolic execution very difficult, user guided symbolic execution will be possible. Another very interesting option is the automatic generation of test sequences. Some work has been done in this field, and a large part of it used attributed grammars as an intermediate form of the protocol from which a test environment or a test pattern generator was derived (lit. [Burkhardt86], [Linn83], [Ural83/84]). The generated test sequences can be used to check the consistency between successive (more detailed) descriptions of a protocol (layer or entity).

2.4 Conclusion

To specify a communication protocol and derive a hardware implementation for it, formal grammars may offer a practical and powerful method. Because of the declarative and action oriented nature of grammars they are likely to be considered intuitive by humans. Because of the known relationship between grammar classes and automata, an implementation method already exists. In this thesis formal grammars will be used a basis for protocol description and implementation, and extensions will be defined to introduce some of the aspects mentioned in par. 2.2.
In the previous chapter, formal grammars were introduced to serve as a basis for the description of protocols. This choice was made mainly because of the expressive properties of grammars, the intuitive (generative) character of grammar definitions and their well known and mathematically provable implementation models.

As a consequence of this choice, it is necessary to consider grammars in more detail and to find out what restrictions must be imposed on general grammars and/or what kind of enhancements must be made to make them more suitable for protocol descriptions and still physically implementable at the same time.

As indicated earlier, others have already done some preliminary work on this topic (ref. [Haas85], [Anderson85a], [Anderson85b]) and have come up with a restriction to context-free grammars and some extensions (attributes and conditional rules). Good arguments for these choices have never been given, and no one has produced a formal definition of the resulting extended grammar or even a related automaton (implementation model) with a mapping function. These definitions are necessary to formally define the semantics of the extended grammars, to give a language definition and to prove that the implementation model is correct.

This chapter formally introduces the concept of grammars and especially context free grammars (CFGs). It shall be argued that the greater expressive power of CFGs is very likely to be an advantage in the description of protocols and for that reason the choice was made to extend CFGs to protocol grammars. A formal definition of attributed context free grammars and protocol grammars will be presented, and a mathematical definition of the language defined by a protocol grammar is given.
3.1 Introduction

Grammars are mathematical models to describe languages whose sentences are recursively enumerable (see [Denning78] and [Lewis81]). They are usually categorized into 4 hierarchical levels or types, known as the Chomsky hierarchy. These are numbered type 0 to type 3 grammars, and they differ in their power to express or define the syntactical properties of languages. Type 0 grammars are the most powerful. They represent the total class of languages with a recursively enumerable set of sentences (the set may have an infinite cardinality). If we denote the set of languages that can be modeled by a type n grammar as LS(n), then LS(n) ⊊ LS(n+1) for n ∈ {0, 1, 2}. In other words: the Chomsky hierarchy is a proper hierarchy.

This means that for every proper n:

- There are languages that can be modeled by a type n grammar, but not by a type n+1 grammar.
- Every language that can be modeled by a type n grammar can also be modeled by a type n-1 grammar.

3.2 The Chomsky hierarchy

3.2.1 Unrestricted (type 0) grammars

Definition 3.1: Grammar.

A grammar G is a 4 tuple $(V_T, V_N, S, P)$ where:

- $V_T$ is a set of symbols (the terminal set). These symbols form the elementary alphabet for the language to be described by the grammar.
- $V_N$ is a set of symbols (the nonterminal set). These symbols do not occur in the language described by the grammar, but serve as intermediate construction symbols to be used in the grammar only. They are also referred to as syntactical classes.
- $V_T \cap V_N = \emptyset$  
- $V = V_T \cup V_N$ the set of all symbols
- $S$ is the axiom (start symbol), $S \in V_N$
- $P$ is a finite set of production rules: $P \subseteq V^* \times V^*$

Production rules are pairs of symbol sequences $(\phi A \psi, \phi \xi \psi)$ with $A \in V_N \land \phi, \psi, \xi \in V^*$. Each pair represents a replacement of a nonterminal (A) appearing in a certain context ($\phi$ and $\psi$) by a sequence of
symbols \( (\xi) \). To visualize this, a rule \( \phi A \psi, \phi \xi \psi \) is usually written as \( \phi A \psi \rightarrow \phi \xi \psi \). The nonterminal replacement according to a production of the set \( P \) is denoted by \( \rightarrow \) and is also known as a derivation step of \( G \). Similarly a repeated application of production rules (i.e. the reflexive transitive closure of \( \rightarrow \)) is denoted by \( \rightarrow^* \) and known as a derivation sequence of \( G \). Formally \( u \rightarrow v \) if and only if there are strings \( \alpha, \beta \in V^* \) and a rule \( (u', v') \in P \) such that \( u = \alpha u' \beta \) and \( v = \alpha v' \beta \).

Starting from the axiom \( S \), new sequences can be generated by repeatedly applying appropriate production rules. Every sequence that can be generated in this way, which consists entirely of terminal symbols is called a valid sequence or sentence of grammar \( G \). Every sequence of terminal symbols, that cannot be derived from \( S \) by applying the rules from the set \( P \) (in any order) a finite number of times is an invalid sequence of \( G \). The language \( L(G) \) described by grammar \( G \) is defined as the set of all sentences of \( G \), i.e.:

\[
L(G) = \{ \xi \mid \xi \in V_T^* \land S \rightarrow^* \xi \}
\]

The main problem for a system implementing a grammar is to determine whether or not a given sequence of symbols is a sentence, i.e. if that sequence can be generated from the axiom \( S \) using a finite number of derivation steps from \( P \). This process is called parsing and a system that can determine if a sequence is a sentence is called a parser. Grammars whose production rules are of the form as described above are the most general. Building an actual parser for type 0 grammars is not only very difficult or even impossible, but for many formal languages it is unnecessary since these can be described by simpler grammars.

### 3.2.2 Context-sensitive (type 1) grammars

A first restriction placed on grammars to reduce parser complexity is this:

For all production rules \( p \) of the general form

\[
p : \phi A \psi \rightarrow \phi \xi \psi ; A \in V_N \text{ and } \phi, \psi, \xi \in V^*
\]

the replacement sequence \( \xi \) may not be the empty sequence:

\[
|\xi| \geq 1
\]

i.e. that the number of symbols on the left hand side (LHS) is not greater than the number of symbols on the right hand side (RHS). An application of a production rule to a sequence will never shorten the sequence. This ensures that there is an
upper limit for the total number of derivation steps required to generate any finite sentence.

Grammars for which the above restriction holds are type 1 grammars. They are called context sensitive grammars, since the LHS of the production rules specify a nonterminal placed in a certain context ($\phi$ and $\psi$) for replacement. The choice for the next production rule to apply depends not only on the current symbol, but also on previous and future symbols (the context). Type 1 grammars are still very difficult to handle and too general for most purposes.

3.2.3 Context-free (type 2) grammars

By introducing another restriction, the complexity of grammars can be further reduced. The context sensitivity is a complicating factor which is often not required. It can be removed by restricting the LHS of every production rule to consist of just one nonterminal symbol:

$$p: A \rightarrow \xi\ ,\ A \in V_N \ and \ \xi \in V^* \ for \ all \ production \ rules.$$  

Grammars whose production rules are all of the above form are called type 2 grammars or context free grammars. Note that the RHS still contains at least one symbol and that so called $\varepsilon$-productions are not allowed. Grammars which contain these $\varepsilon$-productions but which would otherwise be type 2 can always be rewritten in the above form with the possible exception of a single $\varepsilon$-production for the axiom if the empty string is also a sentence. Techniques for parsing $\varepsilon$-extended type 2 grammars ($\varepsilon$-CFGs) have been developed, and therefore $\varepsilon$-productions are usually allowed for type 2 grammars in practice. For convenience in writing, both types will be referred to as context free grammars.

The method for selecting a next production rule to apply when parsing a nonterminal symbol does not depend on other symbols surrounding it (context independent). It can be shown that for a general implementation of CFGs a so called pushdown automaton (PDA) or stack machine is needed (see [Kain72], [Denning78]). This will be explained in more detail later.

Example 3.1 A context-free language and its grammar.

The language consisting of all strings that begin with $n$ a's, followed by $m$ c's and end with precisely $m+n$ b's with $n \geq 0$ and $m \geq 2$ is given by:

$L = \{ a^n c^m b^{n+m} | n \geq 0 \land m \geq 2 \} $
It is not possible to recognize this language with a finite state machine, but it is possible with a pushdown automaton, since there is a context-free grammar describing it. This grammar $G$ is given by:

$$G = ( \{ S, T \} , \{ a, b, c \} , S , \{ S \rightarrow a S b , S \rightarrow c T b , T \rightarrow c T b , T \rightarrow c b \} )$$

It is easy to see that for every accepted $a$ and $c$, a $b$ must also be accepted later (after the last $c$) and that at least 2 $c$'s are required. Hence $G$ defines $L$.

### 3.2.4 Regular (type 3) grammars

Type 3 or regular grammars are a subset of the context-free grammars, obtained by putting yet another restriction on the form of the production rules. The LHS is of the same form as with type 2 grammars, but the RHS may only be a single terminal symbol or a terminal symbol followed by a non-terminal symbol:

\[
p: \quad A \rightarrow aB \quad ; \quad A, B \in \mathcal{V}_N \wedge a \in \mathcal{V}_T \\
p: \quad A \rightarrow a \quad ; \quad A \in \mathcal{V}_N \wedge a \in \mathcal{V}_T
\]

Note: instead of $A \rightarrow aB$ it is also possible to use the form $A \rightarrow B a$. Both formats generate regular expressions.

Regular grammars correspond to finite state machines, also known as finite automata (FA). Every regular grammar defines a regular language and every regular language is defined by some regular grammar. Every regular language can be recognized or generated by a finite automaton. Generally, such a finite automaton will be nondeterministic, but fortunately there exists a straightforward algorithm to convert any NDFA (NonDeterministic Finite Automaton) into a DFA. Any production rule $A \rightarrow aB$ can be interpreted as 'when in state $A$, if the input to the FA is $a$ then the input is accepted and the FA changes its state to $B$'. When there are more production rules for $A$ with the same terminal $a$, then the FA will be nondeterministic.

Any regular grammar can be transformed into a finite state machine. Such an automaton could be nondeterministic. However, by introducing new states, this NDFA can always be transformed into an equivalent DFA (see [Denning78] & [Kain72]). This DFA can be implemented in standard hardware, for example in a PLA. Recent research has resulted in techniques for direct implementation of NDFA's in hardware.

Using this method, the class of regular grammars can be compiled directly into hardware. For other grammars, implementation remains problematic because the corresponding automata are infinite.
3.3 Derivation trees

Information about the derivation of any sequence in a grammar can be stored and visualized in a so-called derivation tree. This is an acyclic directed graph where every node can be reached via exactly one path from the root and is labelled with a symbol. Every node has at most one incoming edge. A root node is a node with no incoming edge. A leaf node is a node without outgoing edges. The other nodes are called intermediate nodes. If node X has an outgoing edge to a node Y, then Y is called a descendant of X, and X is called a parent of Y. For any intermediate node, an ordering is imposed on its descendants (in a graphical representation it is often a left to right ordering).

The derivation tree is restricted in its construction such that for any node labelled N, it can have descendants X₀, ..., Xₖ (given in the imposed ordering) if and only if \( N \rightarrow X₀ \ldots Xₖ \) is a production rule of the grammar. This implies that all nodes with descendants are labelled with nonterminals and that nodes labelled with terminals are (and will always remain) leaves. Furthermore, there is only one root node, and it is labelled with the start symbol. The symbols labelling the leaves in the imposed order give the sequence for which the graph is a derivation tree.

A parser usually constructs a derivation tree in order to find out if a sequence is a valid sentence. A top-down parser would start with the root (start symbol) and add descendants when a rule is chosen (expansion). The process continues until all leaves are labelled with terminals. If the leaf symbol sequence equals the input sequence, it is a valid sentence, otherwise it is either not a valid sentence or a wrong rule was chosen somewhere in the derivation process. A parser has to find out if there exists at least one derivation tree for an input sequence, and if one exists, it should find and construct the tree. If two or more distinct derivation trees can be constructed for any input sequence, the grammar is called ambiguous.

3.4 Context free grammars and pushdown automata

Production rules for an \( \varepsilon \)-CFG (CFG with \( \varepsilon \)-productions) have the general form:

\[
p: A \rightarrow \xi ; A \in V_N \text{ and } \xi \in V^*
\]

An \( \varepsilon \)-CFG can be implemented by a so-called pushdown automaton (PDA). It contains an infinite LIFO stack, a finite state machine and an input reader (see figure 3.1). Depending on the current input and state and the value on the top of the stack,
this machine will compute a next state, optionally accept the input thereby advancing the input reader to the next symbol, and either push a new symbol on the stack, pop the top symbol from the stack or leave the stack as it is. The type of operation and the symbols involved depend on the production rules. If there are more rules for any nonterminal, the automaton becomes nondeterministic (multiple stack operations and next states). Unfortunately, a nondeterministic pushdown automaton (NPDA) can generally not be converted into a deterministic one (DPDA), as is the case with finite state machines. The set of languages recognized by DPDA's is a proper subset of the set of languages recognized by NPDA's.

Initially, the stack contains only the start symbol of the grammar. When the input string is completely processed, the finite state machine is in a final state and the stack is empty, the PDA has accepted the string. If the stack is not empty after the entire input string is read or the finite state machine is not in a final state, the string is not recognized and does not belong to the language. For a proof of the fact that the set of languages recognized by PDA's is exactly the set of context free languages, and for information on how to construct a PDA for a given context free language, see chapter 8 of [Denning78] and [Lewis81] page 112-119.

Example 3.2 A simple CFG and its implementation in a PDA.

The language $L = \{ a^n b^n | n \geq 1 \}$ which consists of all sentences that start with a number of a's followed by exactly the same number of b's, cannot be described by any regular grammar, and hence not be implemented by any DFA.
It is easy to create a PDA that will recognize sentences from \( L \). Consider a PDA whose finite state machine has 2 states, one (\( q_0 \)) in which it can only accept 'a' input symbols and pushes a marker on the stack for every 'a' read from the input, and the other (\( q_1 \)) in which it can only accept 'b' inputs as long as the stack contains at least one marker. As soon as the first 'b' is encountered the PDA changes from its initial state (\( q_0 \)) to the other state (\( q_1 \)) where it remains forever. State \( q_1 \) acts as an acceptance state if and only if the stack is empty. The finite state machine for this PDA is shown in figure 3.2.

It is obvious that only sequences of the from \( a^n b^n \) will leave the stack empty after they have been processed. An example of a grammar \( G \) whose language \( L = a^n b^n \) is:

\[
G = (V_T, V_N, S, P) \text{ where:} \\
V_T = \{a, b\}, \ V_N = \{S\}, \ S = \text{the start symbol and} \\
P = \{ S \to a \ b, \ S \to a \ S \ b \}
\]

The form of the production rules is that of standard context free grammar productions.

![Figure 3.2 The FSM for the PDA that recognizes \( a^n b^n \ (n \geq 1) \).](image)

### 3.5 Protocols and the expressive power of grammars

The actual goal is to describe communication protocols in a grammar, and to compile this mathematical model directly into hardware. An important question is: which type of grammar is needed to describe protocols? Since protocols must be implemented in hardware and/or software, the software is executed by hardware, and the hardware is finite (i.e. there is only a finite amount of memory elements in any system), there is only one conclusion: the number of states in any implementation of any protocol is always finite. This means that the protocol can always be modeled as a finite state machine. In fact, one of the informal protocol specifications of the OSI reference model states that it may never take an arbitrary long time for any implementation to return to some basic state from any state it could possibly be in, inde-
ependent of the behaviour of its environment (stability). Assuming that every action takes a non-zero amount of time, it follows that any sequence of actions which takes a finite amount of time to execute necessarily consists of a finite number of those actions, and this guarantees that there does not have to be an infinite number of states in any implementation. Also, [Harangoz677] has shown that for every implementable protocol there has to be a type 3 grammar describing it.

3.5.1 Separation of data and control space

If protocols would be described using standard grammars (with the entire data space encoded in the control space), such descriptions would be very big. As an example, suppose a protocol requires a DFA with 40 states and a register to hold a variable, whose values may be 1..15. In order to implement this with only a DFA (i.e. using a regular grammar), the register must be eliminated, and its content must somehow be encoded into the state of the DFA. This means that a new state machine has to be created with a state space that is a subset of the Cartesian product of the state space of the old DFA and the register. In this example the new DFA would have up to $15 \times 40 = 600$ states. Any useful protocol has many variables with a much larger range of values. The number of states would grow enormously with protocol complexity and the number of variables used in it. Therefore, it is not practical to describe protocols with standard grammars, unless data space and control space can be separated.

A separation of data and control space requires that both are described separately. The control space is modeled by a standard grammar. The data space, consisting of the variables used in protocols can be modeled very well in attributes, as used in attribute grammars (see section 3.6). Note that the control space is finite for regular grammars and generally unbounded for context-free grammars, because the stack contents are considered part of it. A similar statement holds for the data space, which can grow and shrink as new variables (attributes) are created and deleted dynamically.

3.5.2 Expression of unbounded recursion

Although regular grammars are theoretically sufficient for protocol modeling, it is very likely that context free grammars will lead to easier and more elegant descriptions. Because of the higher expressive power of CFGs, they will be able to express more properties without using attributes and conditions. They also allow more symbols in a rule, which means that states are implicitly coded in positions in rules instead of explicit states represented by nonterminals. Note that CFGs are a proper superset of regular grammars. The essential property that distinguishes the two is
that of self-embedding nonterminals (unbounded recursion). If for any nonterminal \( N \) in \( G \), grammar \( G \) permits:

\[
N \xrightarrow{\ast} \phi N \psi \quad ; \phi, \psi \in V_T^+
\]

then \( G \) has the self-embedding property. It can be shown that every CFG that does not have the self-embedding property can be rewritten in a regular form and therefore describes a regular language.

Example 3.3 Regular grammars versus context free grammars.

In example 3.2 language \( L = \{ a^n b^n \mid n \geq 1 \} \) was introduced. \( L \) is not regular, but it is possible to describe \( L \) in a regular grammar with one attribute and conditions. The attribute is used to count the number of \( a \)'s minus the number of \( b \)'s received. Only when the count equals zero, the machine can reach a final state (condition). However, the CFG implementation needs no extensions at all. Both systems are infinite. The CFG needs an infinite stack and the regular grammar implementation needs an infinite counter.

3.5.3 Unbounded state systems

The previous example highlights another important aspect: finiteness. All finite state systems are definable using regular grammars (RGs). Since CFGs are a superset of RGs, they can define some classes of unbounded state systems. In the future protocols might be modeled as unbounded state systems using these grammars, so that the physical size of the implementation is the only limitation of some of the options these protocols offer (such as the number of simultaneous connections). The protocol becomes finite when it is implemented, as a result of design choices regarding the amount of memory, and not during specification. None of the existing finite state methods would be capable of describing unbounded state protocols. Even for the definition of finite state systems it might sometimes be easier to use a CFG to describe a larger (unbounded) system that completely contains the finite system and to limit the number of states afterwards to the desired set by imposing conditions, than to describe the original system directly using regular grammars (with attributes and conditions).
Example 3.4 An implementation in both CFG and RG with attributes.

Given a regular language \( L = \{ a^n b^n \mid 1 \leq n \leq 100 \} \). This language is clearly finite and can therefore be implemented using a finite state machine architecture (figure 3.3).

![Figure 3.3 Implementation of \( L \) in a finite state machine.](image)

This state machine has 201 states. State \( B_0 \) is reached only after \( n (1\ldots100) \)'s followed by exactly the same number of \( b \)'s. A description in a regular grammar would be:

- \( A_i \to a A_{i+1} \quad ; 0 \leq i \leq 99 \)
- \( A_i \to b B_{i+1} \quad ; 1 \leq i \leq 100 \)
- \( B_i \to b B_{i+1} \quad ; 1 \leq i \leq 99 \)

Thus there are 299 production rules!

The introduction of attributes and conditions can reduce this considerably. Let \( A<n> \) represent nonterminal (state class) \( A \) with an attribute \( n \) whose value is the number of \( a \)'s minus the number of \( b \)'s received thus far. Then

- \( A<n> \to a A<m> \quad ; m = n + 1 \land 0 \leq n \leq 99 \)
- \( A<n> \to b B<m> \quad ; m = n - 1 \land 1 \leq n \leq 100 \)
- \( B<n> \to b B<m> \quad ; m = n - 1 \land 1 \leq n \leq 99 \)

Describes the same system with only 3 production rules (the start configuration is \( A<0> \)).

In example 3.2, the larger system \( \{ a^n b^n \mid n \geq 1 \} \) was given and implemented with the following 2 context free rules:

- \( S \to a b \)
- \( S \to a S b \)

Note the direct self-embedding property for nonterminal \( S \). To limit \( n \) to 100, an attribute and some conditions are introduced for counting:

- \( S<n> \to a b \quad ; n \leq 99 \)
- \( S<n> \to a S<m> b \quad ; n \leq 98 \land m = n + 1 \)
This system requires a maximum of 101 stack positions and is therefore finite! Yet the description is much simpler and more legible than the regular grammar. If it is known that the system will never receive $a^n b^n$ with $n>100$, then the attributes and conditions are not required at all. However, the finite state machine implementation would remain the same under these conditions.

### 3.5.4 Conclusion

The above example clearly demonstrates that context free grammars can considerably reduce the complexity of the description, even with the introduction of attributes. Actually the same principle is being used in other areas as well, such as syntax definition of programming languages. Although no practical program can ever be textually unbounded, the syntax of programming languages is always given in BNF form, which is another notational form of CFGs. This is done because these descriptions are generally shorter and easier to create, to read and to change. In the opinion of the author, the same principle also applies to protocols (since these are also formal languages). For this reason, the protocol grammars are based on context free grammars, rather than regular grammars.

### 3.6 Attributed context free grammars

In the previous section, some arguments were given against the use of regular grammars for the description of protocols. Context free grammars are more powerful, but may lead to nondeterministic automata for their implementation. A restriction to LL(1) grammars can eliminate this problem, but there is no satisfactory way to represent the many 'variables' (context information) of a protocol in standard grammars. It is not feasible to encode these variables in states (or nonterminals), since this would again lead to extremely complex and large descriptions. The more powerful type 0 and type 1 grammars are generally too complex to be handled efficiently (if possible at all) by any physically implementable automaton. Some other way has to be found to accommodate these variables.

If a protocol would be stripped of its variables and only its basic input/output language syntax is considered (control space), then it can be described by a standard grammar. The variable space, representing the total context in which any parsing is to be performed, can then be associated with the productions by means of attributes. This leads to an attributed grammar. In this case, attributed CFGs will be used. Before the definition of an attribute grammar can be given, some general sets must be defined that will be used in the remainder of this and the following chapter.
Attributes correspond to the variables of a protocol. They store the context information in which parsing (symbol acceptance and generation) must take place. Some attributes will have a global nature (they are important to the entire protocol at all times) and some are only relevant for a short time (in a certain phase, or a small part of the protocol), after which they can be deleted. This is analogous to global and local variables in procedural programming languages. To indicate for each attribute if it is local or global, the concept of an attribute scope is introduced.

**Definition 3.2: Attribute scope**

An attribute scope is an element of the attribute scope set $AS$.

$$\text{AS} = \{ \text{global, local} \}$$

Every attribute whose attribute scope is *global* is implicitly known and accessible in every production rule of the grammar and all references in all rules are always to the same single attribute. An attribute whose attribute scope is *local* is newly created whenever a rule is invoked that refers to it, and disposed of when that same rule is completely finished. Therefore, these attributes are only known as long as the rule that caused their creation has not yet completed, and in production rules for nonterminals to which they are passed as attributes.

Local attributes are created, computed and stored in the nodes of the derivation tree during parsing. Since nodes are created dynamically as the derivation tree is constructed, the same holds for these attributes. At every node, only attributes from a certain subtree of the total derivation tree are accessible.

Global attributes can be considered to be stored either in the root of the derivation tree and accessible from every other node or stored outside the derivation tree. Global attributes can be simulated with local attributes, by having them created in the start rule, and passed on to every other rule in the grammar. The concept of globally known attributes is only introduced for convenience of notation. It can considerably reduce the complexity and number of attributes to pass in every rule, and therefore enhance readability of the grammar. It neither introduces anything basically new, nor increases expressive power of the attribute grammar.

In analogy with input and output parameters of procedures in programming languages, attributes of a symbol can be used to pass information along with a symbol or to get information back from a processed symbol. In the first case the value of the
attribute may be used during processing of the symbol, but not changed. In the second case the value is defined (assigned) during symbol processing. To indicate if an attribute acts as an input or output parameter of the symbol processing phase, the attribute mode is introduced.

**Definition 3.3: Attribute mode**

An *attribute mode* is an element of the *attribute mode set* $AM$.

$$AM = \{ \text{inherited, synthesized} \}$$

An attribute that is associated with a symbol $X$ in *inherited* mode must be assigned a value before the symbol $X$ is parsed. While $X$ is being processed the value of the attribute can be used, but it cannot be changed. Such an attribute is used to pass data downward in the derivation tree (from the root towards the leaves).

An attribute that is associated with a symbol $X$ in *synthesized* mode must be assigned a value during parsing of $X$. Therefore, while $X$ is being processed the value of the attribute can be used and changed. Such an attribute passes data upwards in the derivation tree (from the leaves towards the root). The assignment of values to attributes is done through attribute evaluation expressions.

To denote expressions over attributes for context updates and condition testing, the following sets of functions are defined. These will be used in the formal definition of the extensions to context-free grammars.

**Definition 3.4: Boolean and arithmetic functions; arity and range of a function**

Let $W$ be a finite set of values.

The set $BE_n$ of boolean partial functions over $W$ with arity $n \in \mathbb{N}$ is defined by:

$$BE_n = \{ f \mid f: W^n \rightarrow B \}$$

The total set of boolean partial functions over $W$ is:

$$BE = \bigcup_{n \in \mathbb{N}} BE_n$$
Similarly, the set $\mathbb{A}E_n$ of arithmetic partial functions over $\mathbb{W}$ with arity $n \in \mathbb{N}$ is defined by:

$$\mathbb{A}E_n = \{ f | f: \mathbb{W}^n \rightarrow \mathbb{W} \}$$

and the total set of arithmetic functions over $\mathbb{W}$ is:

$$\mathbb{A}E = \bigcup_{n \in \mathbb{N}} \mathbb{A}E_n$$

For any function $f \in \mathbb{A}E \cup \mathbb{B}E$, the arity of $f$ is denoted $\|f\|$, i.e.

$$\|f\| = n \iff ((f \in \mathbb{B}E_n) \lor (f \in \mathbb{A}E_n))$$

and the range of $f$ is denoted $\mathcal{R}(f)$.

Since local attributes are created and deleted dynamically, the total set of attributes changes constantly. It consists of a fixed set of global attributes and a varying set of local attributes which can even contain multiple instances of a single (named) attribute. Not all of them are accessible at any time, and from those only a subset may be changed at a given time. For example, inherited attributes of a symbol may not be changed while that symbol is being processed. To restrict the accessibility of an attribute for retrieving and changing its value, the following definitions are given.

**Definition 3.5: Accessible and alterable attributes**

An attribute $q$ is *accessible* in a certain rule $p$ if and only if at least one of the following conditions hold:

- the attribute scope of $q$ is global.
- the attribute $q$ was created when rule $p$ started (and is therefore a local attribute that has not yet been disposed of).
- the attribute $q$ is associated with the LHS nonterminal symbol of $p$ (and is therefore created in some other rule and passed to $p$ via the attribute passing mechanism).

An attribute $q$ is *alterable* in a certain rule $p$ if and only if the following conditions both hold:

- the attribute $q$ is accessible in $p$, and
- the attribute $q$ is associated with the LHS nonterminal symbol of $p$ in
synthesized mode, or the attribute \( q \) is associated with any RHS symbol of \( p \) in inherited mode or the attribute scope of \( q \) is global.

The informal meaning of accessible is exactly what one would intuitively expect from it: the availability of an attribute to be used in evaluation functions. The informal meaning of alterable is that a new value may be assigned to such an attribute. This should only be allowed for LHS synthesized, RHS inherited and global attributes.

Using the above definitions, the concept of an attributed context-free grammar can now be established formally.

**Definition 3.6: Attributed Context Free Grammar**

An attributed context free grammar \( AG \) is a 6-tuple \((CFG, A, D, AT, VA, SA)\) where:

- \( CFG \) is the underlying context free grammar \((V_T, V_N, S, P)\) as given in definition 3.1 and \( V = V_T \cup V_N \)
- \( A \) is a finite set of names, each of which uniquely defines an attribute. These names will be used to refer to attributes associated with the symbols in production rules and to attributes used in evaluation expressions.
- \( D \) is a finite set of data types, each a subset of some finite set \( W \) of values which is also used in the definition of boolean and arithmetic partial functions (BE and AE).
- \( AT: A \rightarrow D \times AS \) is a function that assigns a data type and an attribute scope to every attribute in \( A \). This mapping determines for every attribute if it is known globally or locally and the set of values it can assume.
- \( VA: V \rightarrow (AM \times D)^* \) is a function that assigns to every symbol \( v \in V \) a (possibly empty) list of \( k \) data types and attribute modes, for some \( k \in \mathbb{N} \) which may depend on \( v \). This mapping defines the number of attributes and their data type associated with every symbol. Note that \( VA \) does not assign actual attributes (names) to symbols. In the set of production rules \( P \), each symbol can be associated with zero or more attributes. All appearances of a given symbol \( v \in V \) in any rule must have the same number of attributes as defined by \( VA \), and the attribute mode and type for each attribute must also
match those given by VA for every appearance of v. By definition \( VA(S) = \emptyset \).

**SA**: {}

To define SA, the following notations are introduced:

For any list \( \xi \), let \( \xi_i \) be the \( i \)th element and \( |\xi| \) be the number of elements (the length) of \( \xi \). Let \( \langle x \in \xi \rangle = (\exists i \in N: 0 \leq i < |\xi| \land x = \xi_i) \).

For all \( p = (N, \xi) \in P \), let \( r_{sp}: [0, \ldots, |\xi|] \rightarrow V \) be a function mapping symbol numbers to symbols defined by:

\[
\begin{align*}
r_{sp}(0) &= N \\
r_{sp}(i) &= \xi_{i-1}; 0 < i \leq |\xi|
\end{align*}
\]

SA is a partial function, defined such that \( SA(p, n) \) assigns to the \( n \)th symbol \((n \in N)\) in rule \( p \in P \) a list of \( k \) attributes \( \alpha \in A^k \) whose attribute modes and types can be found by applying function VA to the corresponding symbol (hence \( k = |VA(r_{sp}(n))| \)), and a set of attribute evaluation functions \( e \). Together they can be represented by a pair \((\alpha, e)\). Each evaluation function \((a, e, l)\) in \( e \) defines the value to be assigned to an alterable attribute \( a \in A \) when the \( n \)th symbol is parsed as the result of evaluating function \( e \in AE \) using values of \( l \in A^* \) as arguments. \( SA(p, 0) \) defines attributes and expressions for the LHS nonterminal symbol, and \( SA(p, i) \) with \( i > 0 \) for RHS symbols. A complex condition, which is defined next must hold for any attributed context free grammar \( AG \).

N.B. In the following expression, text between curly brackets '{' and '}' is comment.

Let \( SA(p, n) = (\alpha_{p}(n), e_{p}(n)) \) for all \( p \in P, n \in N \) for which \( SA \) is defined. Then

\[
\forall p \in P, N \in V_N, \zeta \in V^*: p = (N, \zeta): \{ p \text{ is a production rule } N \rightarrow \zeta \}
\]

\[
\forall n \in N: 0 \leq n \leq |\xi|: \{ n \text{ indexes the symbols in } p, 0 \text{ for } N \}
\]

\[
(\forall (a, e, l) \in e_{p}(n)): \{ (a, e, l) \text{ is an assignment expr. in list } e_{p}(n) \}
\]

\[
\{ a = \text{assignment attribute (destination)} \}
\]

\[
\{ e = \text{function to evaluate (with free variables)} \}
\]

\[
\{ l = \text{a list of attributes, parameters to } e \}
\]

\[
(\exists as \in AS, d \in D: \{ as = \text{attribute scope of } a, d = \text{data type of } a \}
\]

\[
AT(a) = (d, as) \land R(e) \subseteq d \land \|e\| = \|l\| \land
\]

\[
\{ \text{Range of } e \text{ must be subset of type of } a \}
\]

\[
\{ \text{Arity of } e \text{ must match no. of parameters} \}
\]

\[
((as = \text{local}) \Rightarrow \exists i \in N: 0 \leq i < |\alpha_{p}(n)|:
\]

\[
(\alpha_{p}(n))_i = a \land ((n > 0) \Rightarrow (VA(\zeta_{n-1})))_i = (\text{inherited, } d))
\]

\[
\land ((n = 0) \Rightarrow (VA(N)))_i = (\text{synthesized, } d))
\)
{ if a is local, then it must be in $\alpha_p(n)$ and }
{ inherited if $n > 0$, synthesized if $n = 0$ }

\[ (\forall k \in \mathbb{N}: 0 \leq k < |l|: \{ k \text{ indexes the parameters (attributes) for } e \}) \]

\[ (\exists a \in A\mathcal{S}, d \in D: \{ (a, d) = (\text{type, scope}) \text{ of } k^{th} \text{ parameter } \}) \]

\[ (\forall i \in \mathbb{N}: 0 \leq i < |l_k|: \{ \text{if a} \in \mathbb{N}, d \in D: 0 \leq i < |\alpha_p(n)|: \{ i \text{ indexes the symbol attributes } \}) \]

\[ (\forall i \in \mathbb{N}, a \in A\mathcal{S}, d \in D: 0 \leq i < |P_p(n)|: \{ \text{for each attribute of the } n^{th} \text{ symbol, its data } \}) \]

must hold for any attributed context free grammar according to definition 3.6.

The previous condition must hold to guarantee consistency and computability of the expressions when parsing input strings according to AG. Its informal meaning consists of the restrictions already mentioned for VA and SA, the type consistency and the one pass condition for attribute assignment expressions:

Context information regarding symbols that have already been parsed can be stored in inherited attributes and passed down to other parsing routines. Context information obtained by these parsing routines will be returned in synthesized attributes, and can be used to do semantics checking or processing. Even though this seems very powerful, it still is not flexible enough to cover all aspects of protocols. An extended version of the attributed context free grammar, the so called protocol grammar is introduced next.

### 3.7 Protocol grammars

To describe protocols for implementation purposes, the language used for the description must have constructs that are powerful enough to model all characteristics of the protocol which have to be observable in the implementation.
In [Haas85], two characteristics of protocols are mentioned, which are not expressible by the attribute grammars defined earlier:

- **Protocol machines generate outputs while processing their input** (they are language transducers), while normal parsers only accept an input string and (when finished) indicate whether or not the input string belongs to a certain language. The output generation is a part of the communication process to be modeled, and not some side effect. The output language is usually the same as the input language, although not necessarily, and can also be modeled by a grammar. The exact relation between inputs and outputs is described by the rules of the abstract protocol and must now be captured by the grammar. This means that terminals will not only represent inputs, but outputs as well and they should be distinguishable.

- **Protocol implementations can suddenly change their behaviour as a result of certain conditions that arise during a communication.** An example of this is a mechanism called *timeout*, which usually means that the implementation has been waiting for a certain event to occur for more than a predetermined amount of time, after which it will assume that the event will no longer occur. The implementation will then have to perform some special actions to recover from this situation. In a grammar description of a protocol, behaviour is described by sequences of actions modeled as grammar symbols (terminals and nonterminals). The set of rules determines the language that is accepted/generated by specifying an exact relation between input and output symbols, and as such can be viewed as an 'I/O behaviour description'. Unless it is possible to detect (encode) a required behaviour change from (into) the system state and apply the appropriate rules in such states, the behaviour can only change if the rules change, so there must be some way to dynamically change either the production rules themselves or the set of production rules (add/remove entire rules).

Clearly, none of the grammars mentioned before is powerful enough to model these protocol characteristics. Two additional extensions to the concept of attributed context free grammars are now made to eliminate this deficiency. A protocol implementation should be capable of communicating over several inputs and outputs, instead of just one. One such input or output is called a channel. Every channel has a direction that indicates if it is an input or output of the protocol, and a unique identification number by which it can be distinguished from the others. This concept is formalized by the following definition.
**Definition 3.7: Terminal Channel**

A terminal channel is a pair \((c, n) \in \text{ChDir} \times \mathbb{N}\) where \(\text{ChDir}\) is the channel direction set:

\[
\text{ChDir} = \{ \text{input, output} \}
\]

The first element is called the channel direction, and the second is called the channel identification number. Terminal channels represent a property of terminal symbols. A terminal whose channel direction is input represents an input action (the acceptance of a symbol, like normal terminals in standard grammars). A terminal whose channel direction is output represents generation of an output symbol, for which there is no equivalent in standard grammars. The channel identification number is used to represent multiple input and outputs. Every input sequence has a unique number (amongst all inputs), and similarly all output sequences have a unique number. The set of all terminal channels is called TC.

In normal attributed grammars, it is customary to allow terminals to have synthesized attributes only if the terminals are regarded as being accepted (input to the system). Conversely, if the grammar is regarded as a generator all terminals are outputs and the attributes carry information out of the system, hence they should be inherited. For protocol grammars which have both input and output terminals this results in input terminals that may only have synthesized attributes and output terminals that may only have inherited attributes.

**Definition 3.8: Protocol Grammar**

A protocol grammar PG is a 3–tuple \((AG, TA, PC)\) where:

\[
\begin{align*}
\text{AG} & \text{ is the underlying attributed context free grammar} \\
& \text{((} V_T, V_N, S, P), A, D, AT, VA, SA \text{)} \text{ as given in definition 3.6.} \\
\text{TA: } V_T & \to \text{ TC} \\
\end{align*}
\]

\(\text{TA}\) is a function that associates every terminal symbol in the basic underlying context free grammar with a terminal channel. The channel directions partition the terminal set into two disjoint sets, one for input actions and one for output actions. Each of these two sets are then further partitioned into channel sets by the channel identification numbers. Since no two channel sets have a non-empty intersection, multiple input and output sequences (channels) can be represented by a single input and output whose con-
tents are the interleaved versions of the multiple inputs and outputs, such that an original input or output sequence can be retrieved by projecting the combined sequence onto the alphabet (channel set) of that input or output channel.

By definition the following condition must be satisfied:
\[ \forall v \in V_T, \ am \in AM, \ d \in D: \]
\[ (am, d) \in VA(v) \Rightarrow \exists n \in N: \]
\[ ((am = \text{synthesized}) \land (TA(v) = (input, n))) \lor \]
\[ ((am = \text{inherited}) \land (TA(v) = (output, n))) \]

PC: \( P \rightarrow \text{BE} \times A^k \)

PC is a function that assigns a boolean function \( f \in \text{BE} \) with a list of attributes \( \alpha \in A^k \) (where \( k = |\alpha| \)) to every production rule in the basic grammar, whose result is defined as the evaluation of \( f \) using the values of \( \alpha \) as arguments. This set of so called 'enable conditions' dynamically partitions the set of production rules into two disjoint sets, those for which the expression evaluates to true, and those for which it evaluates to false. By definition, only those rules for which the enable condition evaluates to true are considered part of the currently effective grammar and may be used by the parser. The enable conditions may depend on inherited attributes associated with the nonterminal on the LHS of the rule for which the condition is specified, on global attributes and in particular on time, which implies that they have to be constantly re-evaluated during next rule selection. The following condition must hold:

\[ \forall p \in P, \ N \in V_N, \ \zeta \in V^*, \ \lambda \in A^*, \ \varepsilon \in (A \times \text{AE} \times A^*)^*: \ p = (N, \ \zeta) \land \text{SA}(p, 0) = (\lambda, \ \varepsilon): \]
\[ \{ p \text{ is production rule } N \rightarrow \zeta \} \]
\[ \{ \lambda \text{ is the list of attributes associated with } N \} \]
\[ \{ \varepsilon \text{ is the list of assignment expressions for synthesized attributes of } N \} \]
\[ \exists b \in \text{BE}, \ k \in N, \ \alpha \in A^k:\]
\[ (k = |\alpha|) \land \text{PC}(p) = (b, \ \alpha) \land \]
\[ \{ \text{the arity of the rule condition function equals the no. of parameters} \} \]
\[ \forall i \in N: \ 0 \leq i < |\alpha|:\]
\[ \{ i \text{ indexes the parameter attributes of the rule condition} \} \]
\[ (\exists d \in D, \ as \in \text{AS}: \ (AT(\alpha_i) = (d, \ as) \land ((as = \text{local}) \Rightarrow \]
\[ \exists j \in N: \ 0 \leq j < |\lambda|: \ (\lambda_j = \alpha_i \land \text{VA}(N)_j = (\text{inherited, } d)) \}) \} \]
\[ \{ \text{if the } i^{th} \text{ parameter is a local attribute with type } d, \text{ then it must be an} \} \]
\[ \{ \text{inherited attribute of the LHS nonterminal } N \} \]
Protocol grammars according to this definition have enough descriptive power to model both input and output actions and their relation. The partitioning of the production rules also allows for a dynamic set of production rules, and therefore a dynamically changing behaviour. This makes it very easy to model time-outs. In principle the conditions could even allow the underlying context free grammar to be ambiguous, as long as the conditions are such that any conflict can be resolved at parse time by means of restricting choices between rules such that the effective grammar is no longer ambiguous. This higher degree of freedom can also lead to more comprehensible grammar descriptions.

3.8 Example of a protocol grammar

The protocol grammar defined below represents a small one way protocol that transports one packet to another station. It starts by sending the packet (tx_data, rule p1) and then waits for an acknowledge (rectimeack). If the acknowledge is not received within Δt time units, the packet is retransmitted (rule p3) and timing restarted for the acknowledge. Retransmission takes place at most 10 times, after which the protocol gives up (rule p4). In this case the synthesized attribute 'f' in rule p0 is set to true when the protocol finishes, otherwise it is set to false. The function Time is supposed to return a value representing the current time. The function After indicates if the current value of Time is later than its argument.

Let \( F = \{ 0, \ldots, n \} \) for some fixed \( n \in \mathbb{N} \).

\[
\begin{align*}
P &= \{ \text{start} \rightarrow \text{tx} \} \\
&= \{ \text{tx} \rightarrow \text{tx}_\text{data} \text{ rectimeack} \} \\
&= \{ \text{rectimeack} \rightarrow \text{rx}_\text{ack} \} \\
&= \{ \text{rectimeack} \rightarrow \text{tx} \} \\
&= \{ \text{rectimeack} \rightarrow \varepsilon \} \\
\end{align*}
\]

\[
\begin{align*}
A &= \{ \text{cnt}, \text{cntl}, t, f, f1 \} \\
B &= \{ F, B \} \\
\end{align*}
\]

\[
\begin{align*}
\text{AT}(a) &= (F, \text{local}); a \in \{ \text{cnt}, \text{cntl}, t \} \\
\text{AT}(a) &= (B, \text{local}); a \in \{ f, f1 \} \\
\text{VA}(t) &= \emptyset; t \in \{ \text{tx\_data}, \text{rx\_ack}, \text{start} \} \\
\text{VA}(\text{tx}) &= ((\text{inherited}, F), (\text{synthesized}, B)) \\
\text{VA}(\text{rectimeack}) &= ((\text{inherited}, F), (\text{inherited}, F), (\text{synthesized}, B)) \\
\end{align*}
\]
Protocol Grammars

SA(p0, 0) = (Ø, Ø)  
SA(p0, 1) = ( (cnt, f), { (cnt, g0, Ø) } ) with g0 = λ.0  
SA(p1, 0) = ( (cnt, f), { (f, g1, f1) } ) with g1 = λx:Fx  
SA(p1, 1) = (Ø, Ø)  
SA(p2, 2) = ( (cnt, t, f1), { (t, g2, Ø) } ) with g2 = λ.Time  
SA(p2, 0) = ( (cnt, t, f), { (f, g3, Ø) } ) with g3 = λ.false  
SA(p2, 1) = (Ø, Ø)  
SA(p3, 0) = ( (cnt, t, f), { (f, g1, f1) } )  
SA(p3, 1) = ( (cnt1, f1), { (cnt1, g4, cnt) } ) with g4 = λx:Fx+1  
SA(p4, 0) = ( (cnt, t, f), { (f, g5, Ø) } ) with g5 = λ.true  
TA(tx_data) = (output, 0)  
TA(rx_ack) = (input, 0)  
PC(p0) = (g5, Ø)  
PC(p1) = (g5, Ø)  
PC(p2) = (g5, Ø)  
PC(p3) = (g6, (cnt, t)) with g6 = λx,y:F(x < 10 ∨ After(y + Δt))  
PC(p4) = (g7, (cnt, t)) with g7 = λx,y:F(x ≥ 10 ∨ After(y + Δt))

This grammar has on purpose been given according to the mathematical tuple definition to show that for readability purposes another format is very desirable. A so called protocol interface language that allows the same grammar to be described in a much more readable form has been developed and is used in the design system. More detail on this follows in chapter 7.

Rewriting the above protocol grammar gives the following more readable form. Here attributes are written directly following each symbol between brackets. A minus sign (-) indicates inherited mode, and a plus sign (+) indicates synthesized mode. Evaluations are written directly below each rule between straight brackets indexed with the symbol number in the rule (counting from 0 at the LHS) for which the expression has to be computed. The value 0 means that the expressions must be evaluated when all symbols on the RHS have been parsed (so called one pass condition). Conditions for rules are written directly below the evaluations following a meta symbol 'cond.'

Input terminal = { rx_ack }  
Output terminal = { tx_data }  
Nonterminals = { start, tx<-F, +B>, rectimeack<-F, -F, +B> }  
Start symbol = start  
Rules =  

\[
\begin{align*}
\text{start} & \rightarrow \text{tx}<\text{cnt}, +f> \\
& \quad [\text{cnt}:=0] 
\end{align*}
\]
, tx<-cnt, +f> \rightarrow tx\_data \quad \text{rectimeack<-cnt, -t, +f1>}
\[ t := \text{Time} \]
\[ f := f1 \] 0

, rectimeack<-cnt, -t, +f> \rightarrow rx\_ack
\[ f := \text{False} \] 0

, rectimeack<-cnt, -t, +f> \rightarrow tx<-cnt1, +f1>
\[ cnt1 := cnt + 1 \]
\[ f := f1 \] 0
\[ \text{cond:} \; cnt < 10 \land \text{After}(t + \Delta t) \]

, rectimeack<-cnt, -t, +f> \rightarrow \varepsilon
\[ f := \text{True} \] 0
\[ \text{cond:} \; cnt \geq 10 \land \text{After}(t + \Delta t) \]

This format is somewhat similar to that of the defined protocol interface language.

3.9 Derivability and language definition

In this section, the definition of the language L(PG) accepted and generated by a protocol grammar PG = (((VT, VN, S, P), A, D, AT, VA, SA), TA, PC) will be established formally. For context free grammars (VT, VN, S, P), this can be done by defining how a sentential form \( \varphi;n_1\beta_i \) with \( \varphi_i \in V_T^* \), \( n_i \in V_N \) and \( \beta_i \in (V_T \cup V_N)^* \) can be transformed into another one using the rules of the grammar. For protocol grammars this will turn out to be more difficult.

In protocol grammars, attributes are used to store context information that can be used to control parsing by means of conditions on rules. It is therefore not sufficient to denote an intermediate derivation state by just a sentential form, as is the case with normal context free grammars. Instead, a derivation state is now a pair \( <\omega, \rho> \) where \( \omega \) is a sentential form and \( \rho \) is an ‘environment function’ that maps attributes onto values, as defined in the following section.

3.9.1 Operations and data types

The dyadic operator \( \downarrow \) denotes projection of sequences onto an alphabet.
Lists and sequences are constructed using the dyadic concatenation operator: \( \circ \)
Any list \( \zeta \) over \( K \) is either empty, denoted \( \emptyset \), or a concatenation of an element \( \kappa \in K \) and a list \( \zeta' \) over \( K \), denoted \( \kappa \oplus \zeta' \). Concatenation is not commutative, but it is associative: \((\kappa_1 \oplus \kappa_2) \oplus \kappa_3 = \kappa_1 \oplus (\kappa_2 \oplus \kappa_3)\). For convenience in notation, the \( \oplus \) will be omitted whenever this does not lead to ambiguities or mistakes. Also \( \zeta \oplus \emptyset \) and \( \zeta \) are considered semantically equivalent. The reversed list of \( \zeta \) is \( \zeta^R \). By definition \( \emptyset^R = \emptyset \) and \( (\kappa \oplus \zeta)^R = \zeta^R \oplus \kappa \).

For a protocol grammar \( \text{PG} = (((V_T, V_N, S, P), A, D, AT, VA, SA), TA, PC) \) the following data types and functions are required and defined on behalf of the formal language definition:

\[
\begin{align*}
W \cup \{ \bot, T \} &= \text{"Finite set of values, } \bot = \text{unallocated, } T = \text{undefined"} \\
E &= A \times AE \times A^* = \text{"The set of attribute assignment expressions over } W" \\
\Sigma &= \{ (v, \alpha, \varepsilon) \in ((V_T \cup V_N) \times A^k \times E^* \mid k = |VA(v)| \} \\
&= \text{"Unprocessed symbol with attributes and expressions"} \\
OS_j &= \{ t \in V_T \mid TA(t) = \text{"(output, j)"} \} = \text{"} j \text{th Finite output symbol alphabet"} \\
IS_j &= \{ t \in V_T \mid TA(t) = \text{"(input, j)"} \} = \text{"} j \text{th Finite input symbol alphabet"} \\
OS &= \bigcup_j OS_j = \text{"The total combined output symbol alphabet of PG"} \\
IS &= \bigcup_j IS_j = \text{"The total combined input symbol alphabet of PG"} \\
O_j &= \{ (t, w) \in V_T \times W^k \mid TA(t) = \text{"(output, j) \& } VA(t) = k \} \\
&= \text{"} j \text{th output channel alphabet (symbols with attribute values)"} \\
I_j &= \{ (t, w) \in V_T \times W^k \mid TA(t) = \text{"(input, j) \& } VA(t) = k \} \\
&= \text{"} j \text{th input channel alphabet (symbols with attribute values)"} \\
O &= \bigcup_j O_j = \text{"The combined output channel alphabet"} \\
I &= \bigcup_j I_j = \text{"The combined input channel alphabet"} \\
A_G &= \{ a \in A \mid \exists d \in D: AT(a) = (d, \text{global}) \} = \text{"The set of all global attributes in PG"} \\
A_L &= \{ a \in A \mid \exists d \in D: AT(a) = (d, \text{local}) \} = \text{"The set of all local attributes in PG"} \\
\text{ENV} &= \{ p \mid (p: A \rightarrow W) \land \forall a \in A: (\exists d \in D, s \in AS: AT(a) = (d, s) \land p(a) \in d) \} \\
&= \text{"set of attribute environment functions"} \\
\text{Every environment } p \text{ can be partitioned in a global and a local part:} \\
\rho_G: A_G \rightarrow W, \quad \rho_L: A_L \rightarrow W \quad \text{with } \rho_G \cup \rho_L = \rho \\
\text{The function } \Theta \text{ extracts all local attributes from a sequence of stack symbols:} \\
\Theta: \Sigma^* \rightarrow A^* \quad \text{; defined by: } \Theta(\emptyset) = \emptyset \land \Theta(\langle \mu, \alpha, \varepsilon \rangle \oplus \sigma) = (\alpha \upharpoonright A_L) \oplus \Theta(\sigma)
\end{align*}
\]
Changing the value of an attribute $a \in A$ in an environment $\rho \in \text{ENV}$ to $w \in \text{W}$ is denoted by $\rho[a/w]$. The result is a new environment, such that:

$$\rho[a/w](a) = w \quad \text{and} \quad \rho[a/w](b) = \rho(b) \quad ; \quad b \neq a$$

Generalization to lists: $\rho[a \oplus \alpha / w \oplus \omega] = \rho[a/w][\alpha / \omega]$ ; $\alpha \in A^*$, $\omega \in \text{W}^*$, $|\alpha| = |\omega|

Generalization to sets: $\rho[\cdot \wedge / w] = \rho[\cdot / w][(\cdot \wedge \{\varphi\}) / w]$ ; $\varphi \in \Phi$ and $\rho[\cdot / w] = \rho$

Set value substitution changes the value of all attributes in the set to the new value.

Two partial semantic evaluation functions handle expression evaluation:

$\text{EV}: E^* \rightarrow \text{ENV} \rightarrow \text{ENV}$ ; maps assignment expressions with arguments into an environment transformation

$\text{CV}: (\text{BE} \times A^*) \rightarrow \text{ENV} \rightarrow \text{B}$ ; maps a condition expression onto a function that extracts the boolean value of the expression from an environment.

$\text{EV}$ and $\text{CV}$ are defined by:

$$\text{EV}((a, e, \alpha \oplus e)(\rho) = \text{EV}(e)[a / e(\rho(\alpha))]$$

$$\text{EV}(\cdot)(\rho) = \rho$$

$$\text{CV}(b, \alpha)(\rho) = b(\rho(\alpha))$$

### 3.9.2 Expression computability

Semantic evaluation of expressions is not always possible. A precondition for the computability of an expression is that all attributes on which it depends have been assigned a value. Since the order of evaluation is from left to right in the production rule, the precondition is satisfied if (apart from global attributes) expressions only depend on synthesized RHS attributes evaluated for symbols further to the left and LHS inherited attributes. This is generally known as the on~pass condition. Let $p: X_0 \rightarrow X_1 ... X_n$ be a production rule with an attribute assignment such that $i_k$ represents the combination of all inherited local attributes of $X_k$ and $s_k$ is the combination of all synthesized local attributes of $X_k$. The one pass condition states that any inherited attribute of any symbol on the RHS may only depend on attributes that appear in other symbols to its left in the RHS and on $i_0$, and that $s_0$ may depend on $i_0$ and all RHS attributes:

$$s_0 = f_0 (i_0 , s_1 , ... , s_n )$$

$$i_1 = f_1 (i_0 )$$

$$i_k = f_k (i_0 , s_1 , ... , s_{k-1} ) \quad k = 2...n$$

where all $f_i$ ($0 \leq i \leq k$) are computable functions.
Global attributes have not been taken into account here. These are always assumed to have been defined and hence never impose restrictions. The attributes $s_1 \ldots s_n$ must all be stored until the LHS synthesized attribute $s_0$ is computed (this is the last one to compute). However, in general those restrictions are too severe. Computability is also guaranteed if:

$$s_0 = f_0 (i_0, \ldots, i_n, s_1, \ldots, s_n)$$

$$i_1 = f_1 (i_0)$$

$$i_k = f_k (i_0, \ldots, i_{k-1}, s_1, \ldots, s_{k-1}) \quad k = 2..n$$

which means that attribute evaluations may also depend on previous inherited attributes. This condition imposes a restriction on the expressions that are allowed for attribute evaluations.

### 3.9.3 A mathematical reshape of the production rules

The previous definition of PG as a tuple elegantly shows how standard context free grammars were extended to obtain protocol grammars. However, for the remainder of this and the next chapter, it is more convenient to redefine a protocol production rule as a 3-tuple $<b, n, \eta>$ where (see section 3.8 for an example):

- $b \in BE \times A^*$ is a boolean condition expression
- $n \in V_N \times A^* \times E^*$ is the LHS nonterminal with all of its attributes and a list of evaluation expressions for all LHS synthesized attributes, which have to be computed as soon as the RHS of the rule ($\eta$) has been completely parsed.
- $\eta \in ((V_N \cup V_T) \times A^* \times E^*)^*$ is the RHS of the rule, which is a sequence of symbols with associated attributes and evaluation expressions for the inherited attributes.

The set of rules from PG after rewriting into 3-tuples is called PR. It is obtained from PG using a help function $Z$ which combines every symbol with its attributes and evaluation expressions, and which is defined recursively over sequences of symbols.

$$Z: P \times N \times V^* \rightarrow \Sigma^*$$

where

$$Z(r, i, \varnothing) = \varnothing$$

$$Z(r, i, v \oplus v) = <v, \alpha, \varepsilon> \oplus Z(r, i+1, v) \text{ where } <\alpha, \varepsilon> = SA(r, i)$$

Then $PR \equiv \{ <PC(r), Z(r, 0, n), Z(r, 1, v)> \mid r = (n, v) \in P \}$
Note that the above definitions and rewrites do not affect the language defined by the grammar. They only reshape the mathematical format in which the grammar itself is written into a mathematically more convenient one.

3.9.4 Introduction of endmarkers

Thus far, 5 types of symbols have been introduced: processed inputs and outputs (alphabets I and O) and unprocessed input, outputs and nonterminals (combined in stack alphabet Σ). The conversion of an unprocessed terminal to a processed one can be expressed in terms of two functions:

- a 'processor' function μ: (IS ∪ OS) × A* × E* × ENV → I ∪ O and
- an environment mapping γ: (IS ∪ OS) × A* × E* × ENV → ENV

Suppose a protocol grammar PG with one input and one output channel can leftmost derive a sequence where the accepted input is ϕ ∈ I*, the generated output is ψ ∈ O*, and the remaining unprocessed symbol sequence is β ∈ Σ* and that the environment at this point is ρ (the case for multiple inputs and outputs is completely analogous and not of relevance here). Let this grammar configuration GC be denoted by the 4-tuple <ϕ, ψ, β, ρ> ∈ I* × O* × Σ* × ENV. Let DS be a function defining the input/output string pairs derivable from a grammar configuration:

DS: GC → (I* × O*)*

Then terminal processing can be defined in terms of DS as:

DS(<ϕ, ψ, t ⊕ β, ρ>) = DS(<ϕ ⊕ μ(t, ρ), ψ, β, γ(t, ρ)>) ; t ∈ IS × A* × E*
DS(<ϕ, ψ, o ⊕ β, ρ>) = DS(<ϕ, ψ ⊕ μ(o, γ(o, ρ)), β, γ(o, ρ)>) ; o ∈ OS × A* × E*

If there are no more unprocessed symbols, derivation is complete:

DS(<ϕ, ψ, Ø, ρ>) = (ϕ, ψ)

For nonterminals, there is a problem. If a rule is found to expand a nonterminal, then the RHS of that rule must be processed in a new local environment which initially only defines the newly created (and hence undefined) local attributes of that rule and the attributes passed via the nonterminal (which may now have been renamed). After the RHS is completely processed, the current local environment must be restored, except for the attributes that were passed as synthesized to the nonterminal. This sequence cannot be described in the above denotational system without an extension. To extend it, GC is adjusted and a special concatenation operator ⊕GC is introduced, such that the set of all input/output sequence combinations derivable from a grammar configuration A, concatenated with any input/output
sequences derivable from another grammar configuration \( B \) whose environment may depend on how \( A \) was processed, is exactly the set of all sequences derivable from configuration \( A \oplus_{GC} B \). Nonterminal processing can then be defined for a nonterminal \( n = \langle N, \alpha_1, e_1 \rangle \in V_N \times A^{VA(N)} \times E^* \) and a reshaped rule \( <c, \langle N, \alpha_2, e_2 \rangle, \eta> \) as follows:

\[
DS(<\phi, \psi, n \oplus \beta, \rho>) = DS(<\phi, \psi, \eta, \rho_1> \oplus_{GC} <\emptyset, \emptyset, \beta, \rho_3>)
\]

with \( \rho_R = EV(e_1)(\rho) \land \rho_1 = \rho_R[A_L \cup I]\Theta(\eta) \land T] [\alpha_2 / \rho_R(\alpha_1)] \land \{ \text{renaming from } \alpha_1 \text{ to } \alpha_2 \} \)

\[
\rho_3 = (\rho_{R(G)} \cup \rho_{R(L)})(\alpha_1 / EV(e_2)(\rho_2)(\alpha_2)) \quad \text{where } \rho_2 \text{ is such that:}
\]

\[
<\phi', \psi', \emptyset, \rho_2> \text{ is the last configuration encountered in } DS(<\phi, \psi, \eta, \rho_1>)
\]

This concatenation is mathematically inelegant, abuses the type system and looks very artificial. A rather elegant solution is obtained by introducing a special new type of symbol to mark the spot were these environment changes must take place. This symbol is neither a nonterminal nor a terminal and it is not even part of the grammar itself. It will be called an endmarker since it will mark the end of the expansion sequence \( \eta \). The environment change involves the old local environment \( \rho_R \), the two attribute lists \( \alpha_1 \) and \( \alpha_2 \) and the expression list \( e_2 \). Let the symbol itself be denoted by \( \chi \). Then the endmarker type is \( X = \{ \chi \} \times ENV \times A^* \times E^* \times A^* \) and its functionality is defined such that:

\[
DS(<\phi, \psi, n \oplus \beta, \rho>) = DS(<\phi, \psi, \eta, \rho_1> \oplus_{GC} <\emptyset, \emptyset, \beta, \rho_2>) = DS(<\phi, \psi, \eta \oplus \chi, \rho_R, \alpha_1, e_2, \alpha_2 \oplus \beta, \rho_3>)
\]

for the same nonterminal, rule and environment definitions as given above. The endmarker and its parameters form an 'unprocessed' kind of symbol: \( GC \in I^* \times O^* \times (E \cup X)^* \times ENV \) and the definition of \( DS \) for endmarkers follows directly:

\[
DS(<\phi, \psi, \chi, \rho_R, \alpha_1, e, \alpha_2 \oplus \beta, \rho>) = DS(<\phi, \psi, \beta, (\rho_{G(G)} \cup \rho_{R(L)})(\alpha_1 / EV(e)(\rho)(\alpha_2))>)
\]

### 3.9.5 Leftmost derivation steps

The language \( L(PG) \) described by protocol grammar \( PG \) will be defined in terms of leftmost derivations, in contrast to the standard definition where any type of derivation is allowed. Because of the context sensitive parsing introduced by attributes and conditions on rules, the order in which nonterminals are expanded influences the set of derivable sentences (this was not true for plain context free grammars). From a linguistic point of view it would be logical to limit the expansion order to left-to-right, since the grammar directly defines sequences of actions written down in that order. In the next chapter, when automata are discussed, some more detailed arguments for a limitation to leftmost derivations will be given. At this point, the lan-
Language of a protocol grammar is informally defined as the set of leftmost derivable sentences, and all finite prefixes of such sentences. Note that without the inclusion of prefixes, this definition would be consistent with that of standard context free grammars: if all attributes are removed and all conditions set to true, the resulting standard context free grammar defines exactly the same language as it would according to the standard definition, since the derivation order does not influence the language recognized by a standard context free grammar (this property is exactly what the term context free is meant to express). The reason why finite prefixes have to be included is because protocols never end: all derivable sentences are infinitely long and take infinitely long to derive. The goal of the protocol grammars however is to describe any finitely long behaviour that obeys the protocol definition, that is any finite string which can be extended to a sentence.

**Definition 3.9: Input and output configuration.**

An input configuration for a protocol grammar with \( k \) input channels where the channels contain strings \( \alpha_0 \in I_0^*, \ldots, \alpha_{k-1} \in I_{k-1}^* \) is denoted by \(<\alpha_0, \ldots, \alpha_{k-1}>_{GI}\).

Similarly, an output configuration for a protocol grammar with \( m \) output channels where the channels contain strings \( \beta_0 \in O_0^*, \ldots, \beta_{m-1} \in O_{m-1}^* \) is denoted by \(<\beta_0, \ldots, \beta_{m-1}>_{GO}\).

**Definition 3.10: Leftmost sentential form**

A leftmost sentential form for a \( k \)-input \( m \)-output protocol grammar is a sequence \( \phi \beta \) where \( \phi \in (I \cup O)^* \) and \( \beta \in (\Sigma \cup X)^* \) representing accepted input configuration \(<\phi I_0, \ldots, \phi I_{k-1}>_{GI}\), generated output configuration \(<\phi O_0, \ldots, \phi O_{m-1}>_{GO}\) and remaining symbols to process \( \beta \).

**Definition 3.11: Leftmost derivation configuration**

A leftmost derivation configuration \( \delta \) is a pair \(<\omega, \rho>\) in which \( \omega \) is a leftmost sentential form and \( \rho \) is an environment. The set of all leftmost derivation configurations is called \( \Delta \).

**Definition 3.12: Leftmost derivation step**

A leftmost derivation step \( \Upsilon \) is a relation over leftmost derivation configurations: \( \Upsilon \subseteq \Delta \times \Delta \). A leftmost derivation step \( \Upsilon \) is denoted \( \delta_{PG} \) and defined by the following 4 cases:
1. output generation: \( \beta = \sigma \beta' ; \sigma = \langle \tau, \alpha, \varepsilon \rangle \in OS \times A^* \times E^* \)
\[
<\varphi \beta, \rho> \xrightarrow{\delta_{PG}} <\varphi \tau, \rho' (\alpha) > \beta', \rho' > \quad \text{where } \rho' = EV (\varepsilon) (\rho)
\]

If the leftmost symbol to be processed is an output \( \sigma \) consisting of a symbol \( \tau \), an attribute list \( \alpha \) and an expression list \( \varepsilon \), then the environment \( \rho \) is changed into \( \rho' \) by semantically evaluating \( \varepsilon \), \( \sigma \) is removed and an output is generated (on output channel \( j \), where \( TA(\tau) = (\text{output}, j) \)) consisting of the terminal \( \tau \) and its attribute values obtained by applying \( \rho' \) to \( \alpha \).

2. input acceptance: \( \beta = \iota \beta' ; \iota = \langle \tau, \alpha, \varepsilon \rangle \in IS \times A^* \times E^* \)
\[
<\varphi \beta, \rho> \xrightarrow{\delta_{PG}} <\varphi \iota, w > \beta', \rho' > \quad \text{where } \rho' = EV (\varepsilon) (\rho / w)
\]

If the leftmost symbol to be processed is an input \( \iota \) consisting of a symbol \( \tau \), an attribute list \( \alpha \) and an expression list \( \varepsilon \), then \( \iota \) is removed and an input is accepted consisting of the terminal \( \tau \) and its attribute values \( w \) (from input channel \( j \), where \( TA(\tau) = (\text{input}, j) \)). The environment \( \rho \) is then changed into \( \rho' \) by first assigning the input attribute values \( w \) to the attributes in the list \( \alpha \) and then semantically evaluating expressions \( \varepsilon \). If the corresponding channel (\( j \)) is empty, then no progress is made until a symbol becomes available. This symbol must be \( \tau \), otherwise a parse error occurs. This partly expresses the real-time aspect of protocol grammars (interactive construction of input-output strings, as required for protocol systems).

3. expansion: \( \beta = n \beta' ; n = \langle a, \alpha_1, \varepsilon_1 \rangle \in V_N \times A^* \times E^* \)
\[
<\varphi \beta, \rho> \xrightarrow{\delta_{PG}} <\varphi \eta \chi \beta', \rho'' > \quad \text{iff} \quad (\exists r \in P : r = (a \to \nu) \land \eta = Z (r, 1, \nu) \land
\]
\[
\chi = \langle \chi', \rho'_{\omega}, \alpha_1, \varepsilon_2, \alpha_2 \rangle \land
\]
\[
<\alpha_2, \varepsilon_2 > = SA (r, 0) \land
\]
\[
\rho' = EV (\varepsilon_1) (\rho) \land
\]
\[
\rho'' = \rho' [A_L / \bot] [\Theta (\eta) / T ] [\alpha_2 / \rho' (\alpha_1)] \land
\]
\[
CV (PC (r)) (\rho'')
\]

If the leftmost symbol to be processed is a nonterminal \( n \) consisting of a symbol \( a \), an attribute list \( \alpha_1 \) and an expression list \( \varepsilon_1 \), then a rule \( r \) must be found for symbol \( a \) whose transformed right half side, extended with an endmarker is used to replace nonterminal \( n \). First \( \rho \) is changed into \( \rho' \) by evaluating \( \varepsilon_1 \). The local part of \( \rho' \) is stored with the endmarker for later restore when rule \( r \) has been completely processed. Next, \( \rho'' \) is computed from \( \rho' \) by setting all local attributes to be unallocated, then setting all local attributes used in \( r \) to be undefined in value and finally by suc-
cessively assigning the values of the attributes $\alpha_1$ in $\rho'$ to the attributes $\alpha_2$ in the new environment. This effectively creates a new environment where only global attributes are known plus those attributes that are locally defined in the rule $r$, and those that were passed from the current rule (in $\alpha_i$) but which have now been renamed to the new names in $\alpha_2$. The condition expression for the rule $r$ must evaluate to true in the environment $\rho''$. The attribute lists and the list of LHS synthesized attribute expressions are stored with the endmarker to be computed at the end of the rule $r$.

Since $PC(r)$ may depend on time, the set of reachable new derivation states is dynamic in time as well. Let $\Delta_E(t)$ be the reachable set at time $t$. If the original state $<\varphi\beta, p>$ was reached at $t = T_0$, then expansion takes place at $t = T_1$ where:

$$\begin{align*}
|\Delta_E(t)| &= 0 \quad ; \quad T_0 \leq t < T_1 \\
|\Delta_E(t)| &= 0 \quad ; \quad t = T_1 \\
|\Delta_E(t)| &= 0 \quad ; \quad t \geq T_1
\end{align*}$$

i.e. Any allowed expansion must take place as soon as at least one expansion is possible and thereafter no further expansions are attempted from configuration $<\varphi\beta, p>$). This partly expresses the real-time aspect of protocol grammars. A system implementing the grammar will wait as long as no expansion is possible. When multiple expansions are possible, one is chosen nondeterministically and immediately.

4. restoration: $\beta = \chi\beta'; \chi = <\chi', \rho'_\alpha, \alpha_1, \epsilon, \alpha_2> \in X$

$<\varphi\beta, p> \xrightarrow{\delta_{PG}} <\varphi\beta', \rho''>$ where $\rho'' = (\rho'_\alpha \cup \rho''_{\alpha}(\alpha)) [\alpha_1 / \rho''(\alpha_2)] \land \rho'' = EV (\epsilon) (p)$

If the leftmost symbol to be processed is an endmarker $\chi$ consisting of the symbol 'x', an original attribute list $\alpha_1$, a current attribute list $\alpha_2$ and an expression list $\epsilon$ containing evaluations for the synthesized LHS attributes of the current rule, then environment restoration must take place. The expressions $\epsilon$ are semantically evaluated in $\rho$ and the values of attributes $\alpha_1$ in the resulting environment are substituted for attributes $\alpha_1$ in the saved local environment $\rho'_{\alpha_1}$. Together with the environment operations performed during expansion, this effectively defines the attribute passing and local renaming mechanism.

3.9.6 Language definition

Since a protocol grammar defines a language transducer with $k$ inputs and $m$ outputs, the language $L(PG)$ is actually a set of pairs of input-output configurations. The reflexive transitive closure of $\delta_{PG}$ is denoted by $\delta_{PG}^*$. 
Definition 3.13: Language defined by a protocol grammar

For a protocol grammar \( PG = ((V_T, V_N, S, P), A, D, AT, VA, SA), TA, PC) \):

\[
L(PG) = \{ \langle \varphi_i, \varphi_o \rangle | \langle S, \rho_G \rangle \xrightarrow{\delta} \langle \varphi, \rho \rangle \wedge \\
\varphi \in (I \cup O)^* \wedge \beta \in (\Sigma \cup X)^* \wedge \\
\varphi_i = \varphi | I \wedge \varphi_o = \varphi | O \wedge \\
\rho_G = \rho_i [A_G / T] \wedge \rho_i = \lambda \alpha : A . \perp \}
\]

This definition is supposed to match the intuitive meaning of the grammar rules as closely as possible. The different channels are not needed in the language definition, because each symbol belongs to precisely one channel alphabet. Channel information is therefore implicitly present in the definition. The language consists of all finite prefixes of input/output sequence combinations that can be derived from the start symbol and environment \( \rho_G \) using leftmost derivation steps allowed by \( PG \). \( \rho_i \) is an initial environment where all attributes are unallocated. \( \rho_G \) is constructed from \( \rho_i \) by allocating all global attributes (with value 'undefined').

3.10 Modeling of layer hierarchy

3.10.1 Layers, entities and services

Protocol grammars are very suitable for the modeling of layer hierarchies. As mentioned in chapter 1, most bit oriented protocols are specified as a set of hierarchically ordered layers. Processing entities in these layers perform the operations (i.e. provide services) thereby using services provided by lower layers. From within a layer, all lower layers including the transmission medium can be considered a black box capable of performing certain operations. Similarly, each layer entity can forward any service request received from a lower layer which is meant for a higher layer to its adjacent higher layer. The service names, their effects and their parameters are defined in the protocol layer service specification.

The correspondence to protocol grammars becomes clear when layer entities are considered as miniature subprotocols. The external behaviour of each entity can be described by a protocol grammar. Service primitives exchanged by the entities are implemented by symbols generated by one grammar (service user) and accepted by another (service provider). Parameters of the primitives are stored in attributes of suitable types, whose values are always transmitted along with the symbols. The relations between inputs and outputs of a single entity are given by the rules of its gram-
mar (syntactic relations), the expressions over attributes (context changes) and conditions on rules (conditional behaviour). In a somewhat simplified view, an entire protocol can be described by a set of interconnected 'communicating' protocol grammars.

3.10.2 Connected grammars

3.10.2.1 Hierarchical connections

Harangozó has shown how to connect different layers by replacing terminals of one layer with production rules from adjacent layers (see [Harangozó78]). His system was based on regular grammars but can also be applied to protocol grammars. The idea is to use terminals of one layer as start symbols in the next. This way the grammars themselves are ordered hierarchically in the same way as the layers. By actually substituting terminals with production rules from another grammar, a new grammar is obtained that describes the combined behaviour of the two original grammars (although concurrency between the grammars is lost).

The advantage of this specification method is that it allows top down hierarchical specification of protocols layer by layer, and that the same method can be used from the highest level (host commands) to the lowest level (bit patterns for packets). A disadvantage is that the final system description (of the entire protocol) is just an enormously complex single grammar which defines precisely all I/O done by the highest and lowest layers in a sequential manner. This would be very inefficient to implement.

3.10.2.2 Non-hierarchical connections

A different method is followed in this thesis. Grammars for adjacent layers are connected by means of their joint input and output terminals. This method is more versatile because it is not explicitly meant for hierarchical processes but for any combination of communicating processes. Connecting grammars does not create a new single grammar, but keeps the original grammars intact as communicating parallel processes (entities).

Let \{ \text{PG}_i \mid i = 0, \ldots, p \} be a set of connected (and communicating) protocol grammars which together implement a certain protocol. Suppose PG\text{A} and PG\text{B} require communication, for example because PG\text{A} uses a service provided by PG\text{B}. Then the terminal symbol representing that service is defined in both grammars, in this case as output terminal in PG\text{A} and as input terminal in PG\text{B}. The number of attributes,
their order and data type must be the same in both $P_{GA}$ and $P_{GB}$. More generally, every communication terminal has a unique name and is defined in exactly 2 grammars: once as input and once as output. This implicitly defines a protocol grammar interconnection network for point to point communication.

Every protocol grammar $P_{Gi}$ is implemented in a separate automaton. The communications take place over a network of communication channels whose configuration can be extracted from the grammar definitions automatically. This principle will be used for the automatic generation of a protocol engine layout.
Chapter 4

Protocol Automata

The previous chapter introduced and formalized the concept of a protocol grammar, based on standard grammars. First a restriction to context-free grammars was imposed, followed by some extensions to make them more suitable for protocols. These enhancements allow the use of protocol variables (attributes), bidirectional communication on multiple inputs/outputs, and complex context and time dependent behaviour descriptions using conditions on rules. The resulting protocol grammars have enough expressive power to model modern communication protocols.

As indicated in the introduction of chapter 3 (see page 25), it will be necessary to create an implementation model for protocol grammars and a mapping function from these grammars to the model. This must all be done formally, so that the correctness of the mapping function can be mathematically proven.

Although others have worked on describing protocols with grammars like the protocol grammars of chapter 3, no one has yet created an automaton with a construction algorithm and proven it, as has been done for standard grammars in the past. This chapter addresses the problem of finding and proving such a model for the protocol grammars defined in chapter 3.

The theory of languages and automata proves that all 4 categories of grammars can be implemented in certain types of abstract automata, and in particular context-free grammars are implementable by pushdown automata. Protocol grammars were created by extending context-free grammars, and consequently standard pushdown automata are no longer capable of implementing protocol grammars. Similar extensions must also be made to the pushdown automaton. The result is called a protocol pushdown automaton (PPDA). The PPDA will be introduced and formalized in this chapter. It shall be proven that such an automaton can always be constructed for a
given protocol grammar by providing and proving correctness of an algorithm for its construction.

4.1 Pushdown automata

As explained in chapter 3 (see also figure 4.1), context-free grammars (CFGs) can be implemented by pushdown automata. The class of languages recognizable by these machines is identical to the class of languages that can be defined by CFGs. A proof of this can be found in [Lewis81] page 112-119. In particular, to show that any context-free grammar G can be implemented by a pushdown automaton, a construction algorithm for such a machine M for G is given in terms of predefined machine instructions (primitive operations), followed by showing exact correspondence between machine configurations of M and leftmost derivation configurations of G (see also chapter 8 of [Denning78]). The same method will be applied later in this chapter to protocol automata.

A pushdown automaton consists of a finite state machine, an unbounded stack (tape) and an input reader (to sequentially read input symbols from a read only tape). The actions of the machine are described in terms of 3 instructions for stack and input manipulation and a state transition function. During every cycle, the machine makes a state transition and executes one of the 3 instructions.

![Figure 4.1 Pushdown Automaton.](image-url)
Formally, a PDA is a 6-tuple \((Q, \Xi, \Sigma, H, Q_0, Q_f)\) where:
- \(Q\) is a finite set of states
- \(\Xi\) is a finite alphabet of input symbols
- \(\Sigma\) is a finite alphabet of stack symbols
- \(H\) is a set of instructions (the program)
- \(Q_0\) is the set of initial states: \(Q_0 \subseteq Q\)
- \(Q_f\) is the set of final (accepting) states: \(Q_f \subseteq Q\)

The 3 instructions and the corresponding state transitions are denoted by:
- \(\text{q] scan}(t, q')\): applicable when in state \(q\) and the current input symbol is \(t\). Go to state \(q'\) and move input reader to next symbol on input tape.
- \(\text{q] pop}(t, q')\): applicable when in state \(q\) and the top of stack symbol is \(t\). Go to state \(q'\), remove the top symbol \(t\) from the stack and move the stack head back by one position.
- \(\text{q] push}(s, q')\): applicable when in state \(q\). Go to state \(q'\), move the stack head forward by one position and store symbol \(s\) on the stack. Thus the stack head always points to the last symbol written on the stack tape.

Here: \(q, q' \in Q \land t \in \Xi \land t, s \in \Sigma\)

The effects are defined in terms of a so called machine configuration \((q, \varphi, \sigma)\) where \(q \in Q\) is the current state of the finite automaton, \(\varphi \in \Xi^*\) is the input that has already been accepted and \(\sigma \in \Sigma^*\) is the contents of the stack. The configuration describes the entire machine state at some point during execution. The set of instructions defines a relation over machine configurations.

\[
\begin{align*}
\text{q] scan}(t, q') & : (q, \varphi, \sigma) \rightarrow (q', \varphi_1, \sigma) & \text{iff } \varphi_1 \text{ is a prefix of input string } \varphi \\
\text{q] pop}(t, q') & : (q, \varphi, \sigma t) \rightarrow (q', \varphi, \sigma) \\
\text{q] push}(s, q') & : (q, \varphi, \sigma) \rightarrow (q', \varphi, \sigma s)
\end{align*}
\]

Initially, a pushdown automaton \(M\) has a configuration \((q_1, \varnothing, \varnothing)\) where \(q_1 \in Q_i\) and \(\varnothing\) represents an empty sequence. \(M\) accepts string \(\varphi\) if and only if there exists a sequence of instructions whose combined (reflexive transitive) result is:

\((q_1, \varnothing, \varnothing) \rightarrow^* (q_f, \varphi, \varnothing)\) where \(q_1 \in Q_i \land q_f \in Q_f\)

The set of all strings \(\varphi\) accepted by \(M\) is called the language recognized by \(M\). By proper construction of the program \(H\), machine \(M\) can be made to accept exactly those strings that can be derived from a context-free grammar \(G\). Proof of this can be found in several books on the theory of languages and automata (lit. [Denning78]).
and [Lewis81]). The remainder of this chapter will be dedicated to the more complex case of extending the PDA to form a PPDA, a construction algorithm for a PPDA and a mathematical proof of its correctness.

4.2 Extensions to the pushdown automaton

The most important extension defined on context-free grammars is the addition of attributes. An attribute is an instance of an abstract data type. It can represent any of a finite set of values in both scalar and non-scalar form. Attributes are stored in nodes of the derivation tree. Therefore, attributes have to be created when new nodes are added, and they can be deleted when a finished (partial) derivation tree is deleted. Let \( B \) be the set of boolean values: \( B = \{ \text{true, false} \} \). Let protocol grammar \( PG \) be given by \(((V_T, V_N, S, P), A, D, AT, VA, SA), TA, PC)\) as in definition 3.8. The extended pushdown automaton is shown in figure 4.2.

A storage device is needed to hold the values that have been assigned to all attributes known at a specific moment. At any time, it must be possible to assign a new value to any alterable attribute or to retrieve the value of any accessible attribute. Therefore, the storage must be indexed (addressable) and randomly accessible. Its size is generally unbounded, just as the pushdown stack. In terms of automaton components, it can be represented by a tape which is unbounded on the right side and whose alphabet is the finite set of attribute values \( W \). Mathematically, the storage device implements a mapping of attributes to these values, which shall henceforth be called an *environment*.

\[
\text{Environment: } A \rightarrow W \cup \{ \text{undefined, unallocated} \}
\]

The function result 'unallocated' is returned for attributes that are not accessible in the current configuration, and the result 'undefined' is returned for attributes that are accessible but have never been assigned any value since their creation.

Furthermore, a device is required to evaluate the boolean and arithmetic attribute evaluation functions, to assign new values to existing attributes and to dynamically create and delete attributes. This device will be called an *attribute operator*. Mathematically, the attribute operator implements the following mapping:

\[
\text{Attribute Operator: } \text{Environment} \times W^* \rightarrow \text{Environment} \times W^* \times B
\]
The attribute operator is controlled by the finite state machine and it can operate on accessible attributes in the storage and use values present at its inputs. It can generate a sequence of values on its output and return a boolean flag to the finite state machine indicating an attribute error status (equivalent to parse errors: current input does not belong to recognized language). The mechanism for conditions on rules requires no further extensions. The attribute operator can already evaluate the conditions and indicate the result (boolean value) to the finite automaton, which can base its next action on this extra input as well.

![Diagram of Protocol Pushdown Automaton (PPDA)](image)

**Figure 4.2 The Protocol Pushdown Automaton (PPDA).**

The input mechanism has become more complex because terminals can now have associated attributes. The input reader has to accept input symbols as well as their attribute values. The symbols are transferred to the finite automaton as in the standard PDA, but the attribute values are sent to the attribute operator for further processing and/or storage.

An output generator has been added to perform writing of output symbols to an output tape (write only tape). The output symbols are transferred from the finite
automaton to the output generator, while the corresponding attribute values are produced by the attribute operator.

To realize what the computational possibilities of a general protocol automaton are, consider the option of allowing the finite automaton to control the index generator in the attribute storage by means of 3 functions: \texttt{inc}, \texttt{dec} and \texttt{nop} which respectively increment, decrement and leave the index value. Then the protocol automaton of figure 4.2 can simulate any Turing Machine if (and only if) the index value which is used to reflect the Turing head position can assume unboundedly large values. An alternative to unbounded index values (attribute tape positions) is the use of markers on the attribute storage tape to record the position of the simulated head. Reading the tape symbol at the indexed position is then done by first scanning the tape from left to right until the marker is encountered (a finite operation) followed by reading the value stored at that position. This requires operations to move the marker by one position.

The remainder of this chapter will be devoted to the operations of the protocol automaton that are required to implement general protocol grammars.

4.3 Attribute storage management

As explained in chapter 3, local attributes are stored in the nodes of the derivation tree. During parsing, these nodes are created dynamically as the tree is constructed. When a certain subtree is completed and none of its attributes are needed (i.e. will ever become accessible) any longer, it can be deleted and its attributes deallocated. The exact mechanism for the allocation and deallocation of attributes is derived from the definition of accessibility.

Suppose automaton \( M \) is parsing symbols for rule \( p \in P \) and the next symbol to parse is a nonterminal. A choice is made to use rule \( q \in P \) for the expansion. This invocation can be written as \( p \rightarrow q \). Repeating the process for \( q \) and following rules yields an invocation structure. If the automaton is deterministic, there is always at most one applicable rule and the structure will be a linear list, otherwise it will be a tree. For physical implementation, the automaton must be deterministic, so the remainder of this section will concentrate on this case.

Suppose that \( r_0 \rightarrow \ldots \rightarrow r_N \) is the current rule invocation list for a deterministic automaton. When this list is extended with yet another rule \( r_{N+1} \), the only accessible local attributes will be the ones that were passed from \( r_N \) to \( r_{N+1} \) and those that are
completely local in \( r_{n+1} \) and thus newly created while parsing it. These new attributes can be allocated by the time they are actually needed, but separate allocation of every new attribute introduces too much overhead in physical implementations. Therefore another strategy has been chosen: all new attributes for an entire rule are allocated at the same time (when the rule is added to the invocation list). This will increase memory usage, since attributes that are not yet in use have already been allocated. The attributes of previous rules which are still in the invocation list must remain allocated while the current rule is being processed. This is obvious since everyone of these rules will at some time in the future again become the current rule and at that time, its attributes will become accessible. The following informal invariant therefore holds: for any invocation list \( r_0 \rightarrow \ldots \rightarrow r_N \), the attribute store contains all local attributes used in \( r_0 \ldots r_N \) and all global attributes, and none other.

The implication is that deallocation must take place whenever a rule is completely processed (i.e. when the end of a rule is reached). This is the only situation where the invocation list decreases, and to maintain the invariant this is where deallocation of attributes is done. This will of course require a method to determine when the end of a rule has been reached (endmarkers will be used for this).

Since the storage is randomly accessible, there is no implied management system for allocation of indices (addresses) to attributes which are stored in it. However, using the above store invariant, it is easily verified that the allocation and deallocation can be done using a LIFO mechanism (Last In First Out).

### 4.4 Endmarkers

The attribute operator must allocate additional storage when a rule is invoked (non terminal expansion) and deallocate that storage when the end of that rule is reached. Furthermore, as can be seen in section 3.8 the LHS synthesized attributes of a rule must be computed after all RHS symbols have been parsed, i.e. at the end of the rule.

The start of a new rule is easily detected since it is a special action performed by the pushdown controller, but the completion of a rule is not detectable. This is because rules may end with any type of symbol, and there is no way to tell whether a certain symbol on top of the parse stack was put there as the last symbol of a rule during non terminal expansion. Therefore, some additional information must be maintained that allows detection of the end of a rule. Preferably it should also contain information for the operator about the attribute computation. An obvious solution
is to extend each rule with a special symbol, called an *endmarker*. This endmarker is not a symbol from the grammar (terminal or nonterminal). Its purpose is to mark the end of a rule and to control the operator for LHS attribute computation. An endmarker does not have any associated attributes. Its mathematical concept was introduced in section 3.9.4.

When an endmarker is found at the top of the parse stack, the memory deallocation mechanism is activated, which will dispose of all attributes that were allocated for the just completed rule (top segment of attribute stack). Just before deallocation, the operator computes the LHS synthesized attributes, and the endmarker is removed from the stack. In the formal definition of the protocol grammar and in the following formalization of the protocol automaton, the endmarker will be denoted by the symbol $\chi$. Its action is called environment restoration, because it actually restores the environment for the previous rule in the invocation list.

### 4.5 The formal protocol pushdown automaton

In this section the concept of the protocol pushdown automaton will be formalized. Then a construction algorithm will be given to implement any protocol grammar $PG$ and it will be shown that the resulting automaton does indeed recognize exactly the set of sentences that is derivable in $PG$. This section closely relates to section 3.9, where the protocol grammar language was formalized. A number of data types and functions were introduced in that section, which will also be used here. Without explicitly stating this every time, symbols with the same name and capitalization as in chapter 3 will have the same meaning in the following sections. For the remainder of this chapter, let protocol grammar $PG$ be defined by a tuple

$$PG = (((V_T, V_N, S, P), A, D, AT, VA, SA), TA, PC)$$

as given in definition 3.8.

#### 4.5.1 String derivability and language definition

*Definition 4.1:* Protocol automaton

A protocol automaton $M$ with $k$ input tapes and $n$ output tapes is defined as a 10-tuple $(Q, IAS, OAS, U, H, Q_i, Q_f, A_m, W_m, C)$, where:

- $Q$ = A finite set of states
- $IAS$ = A set of $k$ finite input tape symbol label sets \{ $I_{Ao}$, ..., $I_{Ak}$ \}
- $OAS$ = A set of $n$ finite output tape symbol label sets \{ $O_{Ao}$, ..., $O_{An}$ \}
Protocol Automata

$U$ = A finite stack alphabet

$H$ = Program, a finite set of instructions (defined later)

$Q_i \subseteq Q$ = the set of initial states

$Q_f \subseteq Q$ = the set of final states

$A_m$ = A finite set of so-called environment variables (names)

$W_m$ = A finite set of values

$C: IA \cup OA \rightarrow \mathbb{N}$ maps every label from any $IA_i$ or $OA_i$ to an integer:

All sets in $IAS \cup OAS$ are pairwise disjunct. Furthermore, for each $t \in IA_i$ $(0 \leq i < k)$ and for each $t \in OA_i$ $(0 \leq i < n)$, the corresponding tape symbol is a pair $(t, w)$ with $w \in W_m$ and $k = C(t)$. Hence the actual tape alphabets are:

- for the $i^{th}$ input alphabet: $TI_i = \{(t, w) \in IA_i \times W_m \mid c = C(t)\}$ $(0 \leq i < k)$
- for the $i^{th}$ output alphabet: $TO_i = \{(t, w) \in OA_i \times W_m \mid c = C(t)\}$ $(0 \leq i < n)$

Note that these alphabets are all finite.

The total input label set $IA$ and total output label set $OA$ are defined as:

$$IA = \bigcup_{i=0}^{k-1} IA_i \quad \text{and} \quad OA = \bigcup_{i=0}^{n-1} OA_i$$

The total input tape alphabet $\Xi$ and total output alphabet $\Psi$ are defined as:

$$\Xi = \bigcup_{i=0}^{k-1} TI_i \quad \text{and} \quad \Psi = \bigcup_{i=0}^{n-1} TO_i$$

Finally, the set of attribute environment mappings $ENV_m$ is defined as:

$$ENV_m = \{ \rho \mid \rho: A_m \rightarrow W_m \}$$

**Definition 4.2:** Machine input and output configuration

A machine input configuration for a protocol automaton with $k$ input tapes is a set of $k$ sequences $\alpha_0 \in TI_0^*, \ldots, \alpha_{k-1} \in TI_{k-1}^*$ where $\alpha_i$ $(0 \leq i < k)$ represents the contents of the $i^{th}$ input tape, and will be denoted by $<\alpha_0, \ldots, \alpha_{k-1}>_{MI}$.

Similarly, a machine output configuration for a protocol automaton with $n$ output tapes is a set of $n$ sequences $\beta_0 \in TO_0^*, \ldots, \beta_{n-1} \in TO_{n-1}^*$ where $\beta_i$ $(0 \leq i < n)$ represents the contents of the $i^{th}$ output tape, and will be denoted by $<\beta_0, \ldots, \beta_{n-1}>_{MO}$.

**Definition 4.3:** Machine configuration

A machine configuration $\gamma$ of a $k$-input $n$-output protocol automaton $M$ is a 5-tuple $(q, \xi, \psi, \sigma, \rho)$, where:

$q \in Q$ is a state of the FSM.
\[ \xi \in \Xi^* \] is a sequence representing the accepted machine input configuration \(<\xi|T_l_0, \ldots, \xi|T_l_{k-1}>_{M_1}\). For convenience in writing, \(\xi\) shall simply be referred to as the accepted input string.

\[ \psi \in \Psi^* \] is a sequence representing the generated machine output configuration \(<\psi|T_O_0, \ldots, \psi|T_O_{n-1}>_{M_0}\). For convenience in writing, \(\psi\) shall simply be referred to as the generated output string.

\[ \sigma \in U^* \] is the contents of the stack.

\[ \rho \in \text{ENV}_m \] is an attribute environment function, giving the values of all currently defined attributes.

The set of all possible machine configurations is called \(\Gamma\).

**Definition 4.4: Move sequence**

A move of \(M\), denoted \(M \rightarrow\) is a relation over machine configurations, defining the set of possible transitions of machine configurations, such that \(\gamma_1 \rightarrow \gamma_2\) is a move if and only if \(M\), when in configuration \(\gamma_1\) can reach configuration \(\gamma_2\) in a single step.

A move sequence of a protocol automaton (reflexive transitive closure of \(M\)):

\[
(q_0, \xi_0, \psi_0, \sigma_0, \rho_0) \rightarrow^* \cdots \rightarrow (q_N, \xi_N, \psi_N, \sigma_N, \rho_N)
\]

is denoted by \((q_0, \xi_0, \psi_0, \sigma_0, \rho_0) \rightarrow M (q_N, \xi_N, \psi_N, \sigma_N, \rho_N)\)

**Definition 4.5: Language of a protocol pushdown automaton**

A protocol pushdown automaton \(M\) accepts an input string \(\xi\) and transduces it into output string \(\psi\) if and only if:

\[(q_{\text{INIT}} \in \mathcal{Q}, \in, \sigma, \lambda \alpha;A_m \cdot \bot) \rightarrow M (q_f, \xi, \psi, \sigma, \rho)\]

is a move sequence of \(M\).

Allowed initial configurations are all \((q_{\text{INIT}} \in \mathcal{Q}, \in, \sigma, \lambda \alpha;A_m \cdot \bot)\) with \(q_{\text{INIT}} \in \mathcal{Q}_f\).

Final configurations are all \((q_f, \xi, \psi, \sigma, \rho)\) with \(q_f \in \mathcal{Q}_f\).

\[L(M) = \{<\xi, \psi> \in \Xi^* \times \Psi^* \mid \exists q_{\text{INIT}} \in \mathcal{Q}_f, q_f \in \mathcal{Q}_f, \sigma \in U^*, \rho \in \text{ENV}_m: (q_{\text{INIT}} \in \mathcal{Q}, \in, \sigma, \lambda \alpha;A_m \cdot \bot) \rightarrow M (q_f, \xi, \psi, \sigma, \rho)\}\]

Note that the stack does not have to be empty in the final state. This is because all finite prefixes of transducible string sets are also elements of \(L(M)\).

**Definition 4.6: Instructions for the protocol pushdown automaton**

There are 6 different instructions. An instruction is a relation over machine configurations (and therefore defines a move which is generally nondeterministic):

\[\text{Instruction} \subseteq \Gamma \times \Gamma\]
Since each type of instruction is specifically designed to cause a certain effect, these instructions will be given a name which reflects that effect. The name, denotation and effect of the instructions are defined below (result between curly brackets).

\begin{align*}
\rho, q \text{ push} (s, q', \rho') & \quad \text{"push symbol sequence on top of stack"} \\
\{ (q, \xi, \psi, \sigma \oplus s, \rho) \xrightarrow{M} (q', \xi, \psi, \sigma, \rho') \} \\
\rho, q \text{ scan} (<i, w>, q', \rho') & \quad \text{"read input symbol and values and advance input head"} \\
\{ (q, \xi, \psi, \sigma, \rho) \xrightarrow{M} (q', \xi \oplus <i, w>, \psi, \sigma, \rho') \} \\
; \quad \text{iff } i \in IA_i \text{ for some } i \in \{0, \ldots, k-1\} \text{ and the current string on the } \\
\text{ith input tape contains } <i, w> \in T_i \text{ as first symbol.} \\
\{ (q, \xi, \psi, \sigma, \rho) \xrightarrow{M} (q, \xi, \psi, \sigma, \rho) \} \\
; \quad \text{otherwise} \\
\rho, q \text{ test} (<c_0, a_0, \beta_0, q_0>, \ldots, <c_k, a_k, \beta_k, q_k>) & \quad \text{"test a set of conditions"} \\
\text{if } CV(c_0) (\rho[a_0 / \rho(\beta_0)]) \rightarrow \{ (q, \xi, \psi, \sigma, \rho) \xrightarrow{M} (q_0, \xi, \psi, \sigma, \rho) \} \\
\text{fi} \\
\text{with:} & \quad q, q', q_0, \ldots, q_k \in Q; \rho, \rho' \in \text{ENV}_m; c_0 \ldots c_k \in \text{BE} \times A_m^*; \\
& \quad s \in U; \sigma, \sigma_0, \sigma_1, \sigma_2 \in U^*; a_0, \ldots, a_k, \beta_0, \beta_1, \ldots, \beta_k \in A_m^*; \\
& \quad i \in IA; w \in \mathcal{W}_m^*; \xi \in \Xi^*; \pi \in \Psi \text{ and } \psi \in \Psi^* \\
\end{align*}

To express the real time interactive construction of inputs and outputs required for protocols, the input strings on all input tapes can always be extended on the right side with a finite number of new symbols without any intervention or action of M.
For the test instruction, any machine configuration transition whose guard is true can be executed and the other ones cannot. This results in nondeterminism for the general case. If none of the guards is true, the machine configuration is not changed.

4.5.2 Construction algorithm for the automaton

Let PG be a k-input n-output protocol grammar. This section addresses the problem of how to construct a protocol automaton M, such that \( L(M) = L(PG) \).

Obviously, the number of input tapes of M must be k and the number of output tapes must be n. The attributes and their range of values for PG and M must be identical, hence \( A_m = A, W_m = W \) and \( ENV_m = ENV \). The machine and the grammar must also operate using the same input and output alphabets, hence IAS and OAS have to be chosen such that each input channel of PG corresponds to exactly one input tape of M, and that each output channel of PG corresponds to exactly one output tape of M:

\[
OA_i = OS_i \quad \text{and} \quad O_i = TO_i \quad (0 \leq i < n), \\
IA_i = IS_i \quad \text{and} \quad I_i = TI_i \quad (0 \leq i < k),
\]

and \( \forall t \in OA \cup IA: C(t) = |VA(t)| \)

Since M will simulate leftmost top-down derivations, the stack will store unprocessed symbols. In PG, the related alphabet was \( \Sigma \cup X \) whereas in M it is \( U \). Since these must be equal, it follows that \( U = \Sigma \cup X \).

At this point, only the state sets and the instruction set \( H \) remain to be defined. Let

\[
i, j, k \in \{0, \ldots, |P|-1\} \\
n \in \nu, o \in OS \land t \in IS \land w \in \nu^* \\
\alpha, \alpha' \in A^* \land e, e_k \in E^* \land c_k \in BE \land p, p', p'', p''' \in ENV, \\
\{ q_R \} = Q_f \land \{ q_{\text{INIT}} \} = Q, \\
\forall n \in \nu, \oplus \in IS ; o \in OS ; k \in \{0, \ldots, |P|-1\} : q_k, q_{E_k}, q, q_o, q_{T_n} \in Q
\]

(\( q_{E_k} \) is an expansions state using rule \( k \), \( q_{T_n} \) is a test state for nonterminal \( n \))

Furthermore, let the \( k \)th reshaped production rule of PG be given by:

\[
r_k = \langle c_o, \langle n, \alpha_o, e_o \rangle, \eta_k \rangle \quad \text{with} \quad \eta_k \in \Sigma^*
\]

and assume without loss of generality that for any nonterminal of PG, all rules defined for that nonterminal are numbered consecutively.

Expansion sequence

If top of stack equals \( \langle n, \alpha, e \rangle \) then any rule \( r_k = \langle \langle c_o, v_k \rangle, \langle n, \alpha_o, e_o \rangle, \eta_k \rangle \) can be applied if boolean expression \( \alpha \) yields true in the expansion environment when sup-
plied the argument list \( v_k \). Let the set of expansion rules for nonterminal \( n \) be denoted by \( R(n) \). For every nonterminal \( n \) and corresponding set \( R(n) \), \( M \) is constructed such that it contains the following instructions:

\[
\begin{align*}
\rho, q_R & \xrightarrow{rdtos} (\langle n, \alpha, \varepsilon, q_{T, n}, \rho' \rangle) \\
\rho', q_{T, n} & \xrightarrow{test} (\langle \alpha_i, \alpha, q_i, \ldots, \langle \alpha_f, \alpha, q_f \rangle \rangle) \text{ such that } \{ r_i, \ldots, r_j \} = R(n)
\end{align*}
\]

For every rule \( r_k \) defined for any nonterminal symbol \( n \), \( M \) must contain the following instructions:

\[
\begin{align*}
\rho', q_I & \xrightarrow{pop} (\langle n, \alpha, \varepsilon, q_{E, k}, \rho' \rangle) \\
\rho', q_{E, k} & \xrightarrow{push} (\eta_k \oplus \langle \chi', \rho'_{(L)} \alpha, \rho_{(R)} \rangle, q_{R}, \rho_k)
\end{align*}
\]

where:

\[
\begin{align*}
\rho' = EV(\varepsilon)(\rho) \\
\rho_k = \rho'_{(L)} \land \rho'_{(R)} \land \rho'_{(\alpha)}
\end{align*}
\]

Note: As long as none of the test guards is true, the configuration is not changed. This causes the test to be executed again (until at least one guard becomes true).

### Restore sequence

If top of stack contains an endmarker ‘\( \chi' \)’, then an environment restore operation must take place. No condition applies. \( M \) contains the following instruction:

\[
\begin{align*}
\rho, q_R & \xrightarrow{pop} (\langle \chi', \rho'_{(L)}, \alpha, \rho_{(R)} \rangle, \rho'')
\end{align*}
\]

where:

\[
\rho'' = (\rho'_{(L)} \cup \rho''_{(G)})[\alpha / \rho''_{(\alpha)}] \land \rho'' = EV(\varepsilon)(\rho)
\]

### Accept sequence

If top of stack contains an input terminal \( t \), then it must match the next input symbol on the \( i \)th tape, where \( TA(t) = (input, i) \) and hence \( t \in IA_i \). If it does not match, a parse error occurs and operation halts. For every input terminal \( t \), \( M \) contains the following sequence of instructions:

\[
\begin{align*}
\rho, q_I & \xrightarrow{pop} (\langle t, \alpha, \varepsilon, q, \rho \rangle) \\
\rho, q & \xrightarrow{match} (\langle w \rangle, \rho_{(R)}, \rho')
\end{align*}
\]

where:

\[
\begin{align*}
\rho' = EV(\varepsilon)(\rho[\alpha / w])
\end{align*}
\]

### Generation sequence

If top of stack contains an output terminal \( o \), then it must be written to the \( j \)th output tape, where \( TA(o) = (output, j) \) and hence \( o \in OA_j \). No condition applies. For every output terminal \( o \), \( M \) contains the following sequence of instructions:

\[
\begin{align*}
\rho, q_O & \xrightarrow{pop} (\langle o, \alpha, \varepsilon, q, \rho' \rangle) \\
\rho', q_o & \xrightarrow{print} (\langle o, w \rangle, q_{R}, \rho')
\end{align*}
\]

where:

\[
\begin{align*}
w = \rho'_{(\alpha)} \land \rho' = EV(\varepsilon)(\rho)
\end{align*}
\]
Initialization

The machine \( M \) must be initialized by the following instruction:

\[
\rho_{\text{INT}} \text{ push } <S, \emptyset, \emptyset, q_R, \rho_G>
\]

where: \( \rho_{\text{INT}} = \lambda \alpha A . \bot \wedge \rho_G = \rho_{\text{INT}}[A_G / T] \) \{ \rho_{\text{INT}} \) is the initial environment \}

4.5.3 Proof of the construction

This section will provide the mathematical proof that the construction method given in section 4.5.2 is correct.

Theorem 4.1: Given a k-input n-output protocol grammar \( PG \), and a protocol automaton \( M \), constructed according to the method of section 4.5.2. Then \( L(M) = L(PG) \).

Proof:

It has to be proven that \( <\xi, \psi> \in L(PG) \iff <\xi, \psi> \in L(M) \). To do this, it suffices to show that the following assertion holds.

Assertion 4.1:

\[
(q_{\text{INT}}, \emptyset, \emptyset, \rho_{\text{INT}}) \xrightarrow{M} (q_F, \xi, \psi, \beta^R, \rho)
\]

\[
\iff
\]

\[
<S, \rho_G> \xrightarrow{\delta^*} <\varphi \beta, \rho> \wedge \varphi \in (I \cup O)^* \wedge \beta \in (\Sigma \cup \chi)^* \wedge \xi = \varphi I \wedge \psi = \varphi O
\]

with \( \rho_{\text{INT}} = \lambda \alpha A . \bot \) and \( \rho_G = \rho_{\text{INT}}[A_G / T] \) and \( (q_F \in Q_\psi) \) and \( (q_{\text{INT}} \in Q_\psi) \).

In the protocol automaton \( M \), the leftmost derivation configuration \( <\varphi \beta, \rho> \) will be represented by the machine configuration \( (q_R, \varphi I, \varphi O, \beta^R, \rho) \). This is the construction invariant. Hence, every derivation configuration \( \delta \in \Delta \) is represented by exactly one machine configuration \( \gamma \in \Gamma \), and conversely every machine configuration corresponds to exactly one derivation configuration. This bijective relation is denoted by \( \iff \). Assertion 4.1 will now be proven by showing that:

\[
\text{CI: } <\varphi \beta, \rho> \iff M \Gamma \leftarrow \rightarrow (q_R, \varphi I, \varphi O, \beta^R, \rho)
\]

is an invariant of the construction, followed by induction over the derivation steps of \( PG \) and the corresponding moves of \( M \), with special consideration of the initial case.

Let \( \Rightarrow \) represent a move sequence corresponding to one of the 4 basic construction sequence types (accepttion, generation, expansion, restoration). Hence, every \( \Rightarrow \) is also a \( \Rightarrow \). After proving CI invariant, and using the special move sequences \( \Rightarrow \), a stronger version of assertion 4.1 will be proven, namely that the number of leftmost
derivation steps needed to reach a certain derivation configuration equals precisely
the number of \( \text{MC} \) sequences needed to reach the corresponding machine configuration. To prove the invariance of Cl it has to be shown that the following assertion holds for any derivable \( \delta \in \Delta \) and reachable machine configuration \( \gamma \in \Gamma \) and any applicable \( \xi \) and \( \Rightarrow \).

**Assertion 4.2:**

\[
(\delta \xrightarrow{R} \gamma) \Rightarrow (\forall \delta': (\exists \gamma': \gamma \Rightarrow \gamma' \Rightarrow \delta' \Rightarrow \gamma')) \wedge \\
(\forall \gamma': (\exists \delta': \delta \Rightarrow \delta' \Rightarrow \gamma'))
\]

Initially, \( \delta_i = \langle <S, \emptyset, \emptyset>, \rho_G > \) with \( \rho_G = \rho_{\text{INIT}} \) (\( A_g / T \)), \( \rho_{\text{INIT}} = \lambda \in \text{LET} \) which moves by initialization construction (initial push) into \( \gamma_i = \langle q_R, \emptyset, \emptyset, <S, \emptyset, \emptyset>, \rho_G > \). Because \( S \notin \text{V}_n \) it follows from \( \delta_i \) that \( \varphi = \emptyset \) and \( \beta = <S, \emptyset, \emptyset > \). By substitution of these values in Cl, it is easily verified that \( \xi \Rightarrow \gamma_i \). Thus initially CI holds.

To show that CI is invariant, note that in any derivation configuration \( \delta \) of PG at most one of the 4 types of derivation steps is possible (acception, generation, expansion or restoration) and similarly, that in any machine configuration \( \gamma \) at most one of the 4 move types is applicable. By definition of \( \xi \) and the construction mechanism this type is determined by the type of the leftmost unprocessed symbol resp. the symbol on top of the stack.

Suppose \( \delta = \langle \varphi, \rho \rangle \) and \( \gamma = (q_R, \varphi I, \varphi O, \beta^R, \rho) \), i.e. \( \delta \xrightarrow{R} \gamma \)

**Case 1. Output Generation**

\( \beta = \langle \alpha, \alpha, \epsilon > \Rightarrow \beta' \) and \( \gamma = (q_R, \varphi I, \varphi O, \beta^R \Rightarrow \langle \alpha, \alpha, \epsilon >, \rho >) \) with \( \alpha \in \text{OS} \).

By definition: \( \delta \xrightarrow{\text{PG}} \delta' \) where \( \delta' = \langle \varphi <\alpha, \rho' (\alpha) \rangle \beta', \rho' > \) and \( \rho' = \text{EV} (\epsilon) (\rho) \)

By construction, automaton M will contain instructions (pop and print), such that:

\[
\gamma = \langle q_R, \varphi I, \varphi O, \beta^R \Rightarrow \langle \alpha, \alpha, \epsilon >, \rho > \rangle \xrightarrow{\text{pop}} M \langle \text{pop} \rangle \\
\langle q_R, \varphi I, \varphi O, \beta^R, \rho' \rangle \xrightarrow{\text{print}} M \langle \text{print} \rangle \\
\gamma' = \langle q_R, \varphi I, \varphi O \Rightarrow \langle \alpha, \rho' (\alpha) \rangle, \beta^R, \rho' \rangle \quad \Rightarrow \gamma'
\]

denoted by \( \gamma \Rightarrow \gamma' \)

However, since \( \langle \alpha, \rho' (\alpha) \rangle \in O \) and \( (I \cap O = \emptyset) \), \( \gamma' \) can be rewritten as:

\( \gamma' = \langle q_R, (\varphi \Rightarrow \langle \alpha, \rho' (\alpha) \rangle) I, (\varphi \Rightarrow \langle \alpha, \rho' (\alpha) \rangle) O, \beta^R, \rho' \rangle \) and from this follows that

\( \delta' \Rightarrow \gamma' \). Hence Cl is invariant under output generation.
Case 2. Input Acceptance
\[ \beta = \langle t, \alpha, e \rangle \oplus \beta' \text{ and } \gamma = (q_R, q_I, q_O, \beta^R \oplus \langle t, \alpha, e \rangle, p) \text{ with } t \in I. \]

By definition: \( \delta \xrightarrow{p_g} \delta' \) where \( \delta' = \langle q_I, w \rangle \beta', \rho' \) and \( \rho' = EV(e)(p[\alpha / w]) \)

By construction, automaton M will contain instructions (pop and scan), such that:
\[
\gamma = (q_R, q_I, q_O, \beta^R \oplus \langle t, \alpha, e \rangle, p) \xrightarrow{\text{pop}} M
\]
\[
(q_e, q_I, q_O, \beta^R, p) \xrightarrow{\text{scan}} M
\]
\[
(q_R, q_I \oplus \langle t, w \rangle, q_O, \beta^R, \rho' ) = \gamma \text{ iff current input is } \langle t, w \rangle
\]
denoted by \( \gamma \xrightarrow{MC} \gamma' \)

However, since \( \langle t, w \rangle \in I \text{ and } (I \cap O = \emptyset), \gamma \) can be rewritten as:
\[
\gamma' = (q_{R}, \langle \alpha \oplus \langle t, w \rangle \rangle I, (\varphi \oplus \langle t, w \rangle I, \beta^R, \rho') \text{ and from this follows that } (\delta' \xrightarrow{R} \gamma'). \)

Hence CI is invariant under input acceptance.

Case 3. Expansion
\[ \beta = \langle a, \alpha, e \rangle \oplus \beta' \text{ and } \gamma = (q_R, q_I, q_O, \beta^R \oplus \langle a, \alpha, e \rangle, p) \text{ with } a \in V_N. \]

By definition: \( \delta \xrightarrow{p_g} \delta' \) if and only if \( \delta' = \langle q_{\eta}, \chi, \beta', \rho'' \rangle \), such that:
\[
\exists r \in P: (r = (a \to v) \wedge \eta = Z(r, 1, v) \wedge \\
\chi = \langle \chi', \rho'(\_), \alpha, e', \alpha' \rangle \wedge \langle \alpha', e' \rangle = SA(r, 0) \wedge \\
\rho' = EV(e)(p) \wedge \rho'' = p'[A_k / \bot][\Theta(\eta) / T] [\alpha' / \rho'(\alpha)] \wedge \\
CV(C(r))(\rho'')
\]

By construction, automaton M will contain a set of instructions (rdtos, test, pop and push), such that:
\[
\gamma = (q_R, q_I, q_O, \beta^R \oplus \langle a, \alpha, e \rangle, p) \xrightarrow{\text{rdtos}} M
\]
\[
(q_{R_{a}}, q_I, q_O, \beta^R \oplus \langle a, \alpha, e \rangle, p') \xrightarrow{\text{test}} M
\]
\[
\gamma'' = (q_{R_{a}}, q_I, q_O, \beta^R \oplus \langle a, \alpha, e \rangle, p')
\]
\[
\exists r_k \in PR: r_k = \langle c_k, \langle a, \alpha, e \rangle, \eta_k \rangle \wedge CV(c_k)(\rho'[\alpha_k / \rho'(\alpha)]) \xrightarrow{\text{pop}} M
\]
\[
\gamma' = (q_{R_{a}}, q_I, q_O, \beta^R \oplus \langle \chi', \rho'(\_), \alpha, e', \alpha' \rangle \wedge \eta_k, \rho_k) \xrightarrow{\text{push}} M
\]
\[
\gamma = (q_R, q_I, q_O, \beta^R \oplus \langle \chi', \rho'(\_), \alpha, e', \alpha' \rangle \wedge \eta_k, \rho_k) \xrightarrow{\text{push}} M
\]
\[
\exists r_k \in PR: r_k = \langle c_k, \langle a, \alpha, e \rangle, \eta_k \rangle \wedge CV(c_k)(\rho'[\alpha_k / \rho'(\alpha)]) \wedge \\
\rho_k = p'[A_k / \bot][\Theta(\eta_k) / T] [\alpha_k / \rho'(\alpha)]
\]

and thus \( \gamma \Rightarrow_{MC} \gamma' \).

For \( \gamma' \), the condition \( CV(c_k)(\rho'[\alpha_k / \rho'(\alpha)]) \) is clearly equivalent to \( CV(c_k)(\rho_k) \).

Since the set of reshaped rules PR was defined such that:
\[
\exists r \in P: r = (a \to v) \wedge \eta = Z(r, 1, v) \wedge \langle \alpha', e' \rangle = SA(r, 0) \iff \\
\langle C(t), \langle a, \alpha', e' \rangle, \eta \rangle \in PR
\]
it follows immediately that for any \( \delta' \) reached by an expansion step using a rule \( r \in P \) as above, there is a move sequence reaching \( \gamma' \) defined for some reshaped rule \( r_k \in PR \) as above, and for any \( \gamma' \) reached by a move sequence using a reshaped rule \( r_k \in PR \), there is a leftmost derivation step reaching \( \delta' \) defined for some rule \( r \in P \) such that \( r_k \) is the reshaped rule \( r \) and thus \( \alpha' = \alpha_k, \epsilon' = \epsilon_k, \rho'' = \rho_k \) and hence \( \delta' \overset{R}{\rightarrow} \gamma' \).

**Case 4. Restoration**

\[ \beta = x \oplus \beta' \text{ and } \gamma = (q_R, qI, qO, \beta^R \oplus \chi, \rho', \alpha_1, \epsilon, \alpha_2, \rho) \]

with \( x = \chi, \rho', \alpha_1, \epsilon, \alpha_2 \in X. \)

By definition: \( \delta, \delta' \) where \( \delta' = <\phi', \rho''> \) and \( \rho''' = EV(e)(\rho) \)

and \( \rho'' = (\rho'_{(U)} \cup \rho'''_{(V)})[\alpha_i / \rho'''(\alpha_j)] \)

By construction, automaton \( M \) contains a pop instruction, such that:

\[ \gamma = (q_R, qI, qO, \beta^R \oplus x, \rho', \alpha_1, \epsilon, \alpha_2, \rho) \overset{\text{pop}}{\rightarrow} M \]

and therefore \( \gamma \overset{\text{MC}}{\rightarrow} \gamma' \) and \( \delta' \overset{R}{\rightarrow} \gamma' \).

This concludes the proof of assertion 4.2.

Combination of assertion 4.2 and the fact that Cl holds initially, leads to the following assertion:

**Assertion 4.3:**

\[(\forall \delta \in \Delta, k \in \mathbb{N}: \delta \overset{\Delta}{\rightarrow}_k \delta: \exists \gamma \in \Gamma: (\gamma \overset{\text{MC}}{\rightarrow}_k \gamma) \wedge (\delta \overset{R}{\rightarrow} \gamma)) \wedge \]

\[(\forall \gamma \in \Gamma, k \in \mathbb{N}: \gamma \overset{\text{MC}}{\rightarrow}_k \gamma: \exists \delta \in \Delta: (\delta \overset{\Delta}{\rightarrow}_k \delta) \wedge (\delta \overset{R}{\rightarrow} \gamma)) \]

**proof:**

The case for \( k=0 \) is trivial. Now assume that assertion 4.3 is true for \( k=n \) (\( n > 0 \)).

Suppose there is a \( \delta \), such that \( \delta \overset{\Delta}{\rightarrow}_{n+1} \delta \). Then there must be a \( \delta_n \), such that \( \delta \overset{\Delta}{\rightarrow}_n \delta_n \)

\( \delta_n \overset{\Delta}{\rightarrow} \delta \) and a \( \gamma_n \) such that \( \gamma \overset{\text{MC}}{\rightarrow}_k \gamma_n \) and \( \delta_n \overset{R}{\rightarrow} \gamma_n \). Assertion 4.2 then guarantees that there also is a \( \gamma \) such that \( \gamma_n \overset{\text{MC}}{\rightarrow}_{n+1} \gamma \). Conversely, suppose there is a \( \gamma \) such that \( \gamma \overset{\Delta}{\rightarrow}_{n+1} \gamma \). Then there must be a \( \gamma_n \), such that \( \gamma \overset{\text{MC}}{\rightarrow}_n \gamma_n \overset{\text{MC}}{\rightarrow} \gamma \) and a \( \delta_n \), such that \( \gamma_n \overset{\text{MC}}{\rightarrow}_n \delta_n \) and \( \delta_n \overset{R}{\rightarrow} \gamma_n \). Assertion 4.2 then guarantees that there also is a \( \delta \), such that \( \delta \overset{\Delta}{\rightarrow}_n \delta_n \) and \( \delta_n \overset{R}{\rightarrow} \gamma_n \). Assertion 4.2 then guarantees that there also is a \( \delta \), such that \( \delta \overset{\Delta}{\rightarrow}_n \delta \) and \( \delta \overset{R}{\rightarrow} \gamma \). Hence assertion 4.3 is also true for \( k=n+1 \). By induction, assertion 4.3 is then true for any \( k \in \mathbb{N} \).
Since assertion 4.3 is stronger than 4.1, it implies and proves the latter and thereby theorem 4.1. Hence \( L(M) = L(PG) \) and the correctness of the construction algorithm is proved.

At this point a formal implementation model for protocol grammars has been presented, called the protocol pushdown automaton. A formal definition of the language accepted/generated by such an automaton is given, analogous to that given in chapter 3 for protocol grammars. From these definitions a mapping function (presented here as a construction algorithm) followed almost directly. This one way mapping allows any protocol grammar to be implemented by a protocol pushdown automaton, such that both define the same language. The correctness of this mapping has been proven.

The conclusion is that it is now possible to create an implementation for any system described by a protocol grammar. However, it should be noted that the protocol pushdown automaton is a very abstract implementation model. Many of its operations are described only as highly abstract mathematical (symbolic) functions, for which it is not directly clear how to implement them physically. Furthermore, the automaton is in general nondeterministic (depending on the grammar) and requires an unbounded amount of memory.

The first problem will be addressed in chapter 5, where a physical implementation for the protocol pushdown automaton is presented. The latter two are discussed in the next section, where some solutions shall be presented as well.

### 4.6 Problems regarding physical implementations

Pushdown automata are capable of implementing infinite state systems. This is only possible if the automaton itself is also infinite. In case of the standard PDA this is 'achieved' using an unbounded LIFO stack. In the case of the PPDA, there is an unbounded stack and an unbounded attribute storage. Such an automaton is not of any practical use, since it cannot be implemented. Even the very simple example 3.2 from the previous chapter is not physically implementable. However, since protocols are really finite state systems, it is expected that this property will guarantee a maximum stack size and attribute storage size. In this section it will be argued that for so called 'stable' protocols, this is indeed the case.
4.6.1 The stability property of protocols

In chapter 2 the criteria for global correctness of protocols were stated. A correct protocol has the following properties: liveness, free of deadlock, stability and completeness. Especially the stability property is a very interesting one: every protocol should return to some basic state (or set of basic states) within a finite amount of time, independent of the behaviour of the environment. In the protocol pushdown automaton, the state consists of the FSM state, the environment and the stack contents. Many of these states are equivalent, although they appear in a different context. If two states A and B differ only by the values of a few variables which cannot influence the syntactic analysis (i.e. cannot have any effect on enable conditions of rules) then A and B are considered to be in the same state class. A state class is a set of states each with the same FSM state, stack contents and subset of the environment. Two representations of the same protocol state must have the same stack contents, since the stack represents the future behaviour of the automaton (remaining actions to do) from the given protocol state. The stability property is now replaced by a new (generally weaker) version: return to a basic state class within finite time.

**Theorem 4.2:** A stable protocol can be implemented using a bounded parse stack.

**Proof:**
Suppose every step of the physical PPDA takes a positive (non-zero) finite amount of time. Since the basic state class must be reached within finite time from start up, it can only be reached within a finite number of actions. Because every rule of a protocol grammar has a finite length, the actions of the PPDA will all be finite (both in number of steps and duration). Therefore, there must be an upper bound \( U_B \) for the stack size of the basic state class. Now assume that during execution the stack has grown to a size \( U_E > U_B \). To return to the basic state class will take at least \((U_E - U_B) \cdot \tau_{\text{pop}} \) time units, where \( \tau_{\text{pop}} \) is the minimum amount of time required to execute a \textit{pop} instruction (i.e. to pop a symbol from the stack). If this should occur within at most \( K \) time units, then it follows that \( U_E < K \cdot \tau_{\text{pop}}^{-1} + U_B \). Since \( K \) is a finite value from the specification and \( \tau_{\text{pop}} \) is determined by the implementation (but a fixed constant), there is an upper bound for \( U_E \).

**Theorem 4.3:** A stable protocol can be implemented with a bounded attribute store.

**Proof:**
The parse stack contains the unparsed parts of all rules in the invocation list (stored in LIFO order, see section 4.3). Since the parse stack is upper bounded for stable protocols, and every rule in the invocation list consumes at least one stack position
(for the endmarker), the number of production rules in the invocation list is also upper bounded. By definition, both the number of local attributes for each rule and the total number of global attributes are finite. From the attribute store invariant, it follows directly that the total number of allocated attributes is upper bounded. Since all attributes have a finite range of values, the total attribute storage size will also have an upper bound.

4.6.2 Protocol grammars and stability

A stable protocol with finite attribute ranges can be implemented in a finite protocol pushdown automaton. The size of the required parse stack and of the attribute memory are basically computable from the grammar by analytical methods (manually done, usually requires some ingenuity) or by some automated method such as symbolic execution. One question has remained unanswered: is a protocol grammar based implementation stable?

4.6.2.1 Endless loop constructs

Assume that the original abstract protocol specification is stable. The question then becomes: does a protocol grammar description introduce instability? The answer to that question is unfortunately positive. Protocol grammars as defined here can introduce a kind of instability but this can be prevented by a few restrictions. The instability is a direct result of the lack of looping constructs in generative grammars. In fact, any cycle has to be created by tail recursion in the form: \( S \rightarrow^* \omega S \). All rules appended to (but not removed from) the rule invocation list between two successive occurrences of \( S \) on top of the stack will be removed until the second \( S \) is completely reduced to \( \epsilon \). But if the loop is endless \( S \) can never be completely reduced and those rules will never terminate. For every pass through the loop an additional endmarker will be left on the stack for every involved production rule, and similarly all attribute memory for these rules will remain allocated. Any physical implementation would quickly run out of parse stack and attribute memory.

4.6.2.2 Limiting right recursion in protocol grammars

This can be prevented by imposing a limitation on the rules of the protocol grammar and by special treatment of direct right recursive rules as follows:

1) Unlimited indirect right recursion (involving at least 2 rules) is not allowed. Thus constructs such as \( S \rightarrow^* \omega \rightarrow^* \varphi S \) are not allowed (\( \rightarrow^* \) denotes 1 or more applications of a rule to expand a nonterminal) unless the loop is guaranteed to terminate after a finite number of executions. If recursion depth is
limited to some finite value using conditions on rules then the loop will terminate and the stack will not grow infinite.

2) Unlimited direct right recursion (involving a single rule) is allowed, but the recursion nonterminal may not have any associated attributes or attribute evaluation expressions (not even for global attributes). Such a construct is given by \( S \rightarrow \varphi S \). Here \( \varphi \) is any arbitrary sequence of symbols (terminals and nonterminals, except \( S \)) representing the body of the loop. Since \( S \) has no (synthesized) attributes, its expansion cannot change the local environment of the current invocation. Furthermore, since the expression list for the endmarker at the end of this rule is empty, the endmarker's only purpose is deallocation of local attributes. Because \( S \) has no attributes, this can also be done before \( S \) is re-expanded. The endmarker can then be omitted entirely.

4.6.2.3 Stable construction of endless loops

The limitations mentioned in the previous section do not restrict the creation of more complex loops. The following method of 3 sequentially executed steps (in the given order) can be used to create a stable endless loop using direct right recursion:

- divide the behaviour that should be repeated in \( n \) sequential behaviour parts named \( A_1 \ldots A_n \) \( (n \geq 1) \). Define \( A_1 \ldots A_n \) as nonterminals. Each of these behaviours can again be loops with undetermined termination characteristics.
- define the I/O behaviour of the loop components \( A_1 \ldots A_n \) using a set of production rules for these nonterminals.
- define a new unique nonterminal \( S \) without attributes and add the following rule: \( S \rightarrow A_1 \ldots A_n S \). Conditions on this rule determine whether repetition is endless or not. Produce \( S \) wherever the loop is required.

This construction is stable. None of the nonterminals \( A_1 \ldots A_n \) can ever derive \( S \).

4.6.3 Elimination of nondeterminism

The standard pushdown automaton is nondeterministic in the general case. The same holds for the protocol pushdown automaton. It is introduced by expansion actions (nonterminal on top of stack) where a machine configuration can be transformed into any of a set of new machine configurations. A complete physical implementation would have to maintain all of these possibilities. For every nonterminal the number of configurations could grow. This results in a tree structure of possible configurations. If an input cannot be processed in a certain subtree, that subtree can be deleted.
If only one possible next configuration would be chosen according to some criterion, a wrong choice may be made causing the input not to be recognized. A physical machine that emulates a nondeterministic PDA would be very complex and have to perform (in the worst case) a rapidly growing number of computations for each move, which is of course both impractical and undesired (performance decreases with complexity). Therefore the nondeterminism has to be eliminated by putting restrictions on the grammar.

4.6.3.1 Deterministic parsing strategies

From the formal definitions it is clear that if the set of configurations reachable by expansion of a nonterminal contains only one element at the time the expansion takes place, the whole automaton will be deterministic. The set consists of all rules for the nonterminal to be expanded, whose condition evaluates to true. The number of elements in it can be reduced by the following 3 means:

- Reduce the number of rules for any given nonterminal in the grammar
- Put stronger conditions on rules
- Add another selection mechanism to discard rules in real time

Since different rules for a nonterminal usually represent execution alternatives, it is not always possible to eliminate rules by rewriting the grammar. It can help sometimes, but it is not a generally useful method. Conditions form a very powerful mechanism of making selections between rules. However, it is not always possible to express a selection in terms of conditions over attributes. Sometimes all alternatives are allowed and selection of a correct rule must be left to some other mechanism. An often used mechanism is that of the so called look ahead sets. The standard parser (pushdown automaton) contains a database that stores for every rule \( r \) \((r: N \rightarrow a)\) the set of terminal symbols that can appear as the leftmost symbol in any possible sequence derived from \( N \) by starting with rule \( r \). This stored set for every rule is called its look ahead set. Before selecting a rule, the parser will first read the next input terminal, then search in the database (lookup table) to find which rule can derive the given terminal, and apply only that rule. For a 1-input automaton this parsing mechanism is deterministic if the look ahead sets of all rules for any given nonterminal have empty pairwise intersections (see [Aho86]). For \( k \)-input automata, a deterministic choice between tapes must also be made if more than 1 tape delivers a symbol that can be accepted in some configuration. A possibility for this is to use the temporal order of the symbols (arrival time) by timestamping input symbols when they are put on the tape. The timestamps are only used when a choice between tapes must be made. Unfortunately, these methods are only applicable for choosing
between rules that derive input terminals. Output terminal deriving rules must be controlled using conditions. In contrast to normal parsers, the look ahead mechanism alone does not have to guarantee nondeterminism. Conditions on rules can further eliminate conflicts between rules which derive sequences with the same input terminal as the leftmost symbol.

4.6.3.2 LR parsing techniques

Every language accepted by some deterministic pushdown automaton is called a deterministic context-free language (DCFL). The most general set of deterministic grammars is the set of LR(k) grammars. An LR(k) parser scans the input from left to right and constructs a rightmost derivation tree using at most k look ahead symbols. From [Denning78], page 434-437:

- Every LR(k) grammar describes a DCFL
- Every DCFL has an LR(1) grammar describing it which means that nondeterminism can be eliminated if the grammar is LR(1). In the most general case, this is indeed a restriction because there are context-free languages for which no LR(1) grammar exists.

The problem with LR parsers is that they make rightmost bottom-up derivations, whereas the language definitions of both the protocol grammar and the automaton were based on (more intuitive) leftmost derivations. LR parsers would require a redefinition of language and another construction method automaton. One might ask if such a definition is actually possible, since there are 3 categories of problems:

1) **Attribute evaluation problems.** LR(k) parsers only allow attribute evaluation expressions for the LHS attributes (i.e. associated with the whole rule and not with every separate symbol). This is because such a parser does not have information about which rule the current symbol belongs to. This is undecidable until all RHS symbols of a rule have been input (and pushed on the parse stack) and the following k input symbols are known. At that moment the parser can recognize which rule was used and a reduce action takes place to replace the pushed sequence with the LHS nonterminal. Since reduce states are the only ones where the relation of the input string to the grammar is known, they are also the only states in which any attribute evaluation expressions from the grammar can be computed.

2) **Output terminal problems.** These symbols may appear everywhere in the production rules and represent output actions. How does the automaton know when to generate an output? The entire LR parsing mechanism is
based on the principle of first accepting a string and then trying to recognize which rules were used to construct it. This principle will not work for outputs, especially when the I/O strings are constructed interactively between two communicating partners, as is the case with protocols.

3) **Rule condition problems.** Conditions on rules have to be evaluated when a rule is started (i.e. when its first RHS symbol is parsed). This is not possible with LR parsers as rules are detected after they have already been completely parsed.

The first category cannot be eliminated without rewriting the grammar rules so that every symbol with inherited attributes is the last symbol of a rule. Because it is expected that many symbols will have inherited attributes, the resulting grammar will most likely be inefficient (many very short rules). The second category can only be resolved by imposing severe restrictions on the grammar form (output terminals are associated with the end of production rules and cannot appear at other places in any rule). This restriction allows generation of output only when a reduce action takes place. The third category of problems cannot be solved elegantly. It either requires an awkward redefinition of the semantics of conditions on rules such that a condition must hold for a reduce action to be enabled, or it requires a backtracking parsing method which is evidently bad for real-time performance.

4.6.3.3 Predictive parsing techniques

Predictive parsers are top-down parsers. They start with the start symbol and try to apply production rules in such a way that the input string is accepted. Since their principle is based on selecting a rule, and to apply it to a nonterminal on the parse stack, it is always known which rule is current. That is why they are called predictive parsers. It is obvious that they do not have any of the problems that LR parsers have. Predictive parsers exist for LL(k) grammars, which are a subset of the LR(k) grammars. The theory of operation of LR and LL parsers is explained in [Fisher88] and [Aho86]. One specific difference between LR and LL is that for LL parsers the parse stack contains unfinished parts of invoked but uncompleted rules, representing expected future behaviour (inputs/outputs) of the machine. Input terminals on the stack have to match input symbols, output terminals indicate the generation of output symbols and nonterminals must be expanded using one of their rules. The choice for a rule can be made as soon as the next k input symbols are available (k symbol look ahead) in the input queue. Protocol grammars will be restricted to ε-LL(1). The selection mechanism for a rule as expansion of a nonterminal is deter-
ministic and based only on the current parser state and the next input symbol. This parsing technique is very similar to that of the pushdown automaton.

4.6.4 Conclusions concerning implementations

The problem of unbounded memories that would be required for a physical implementation are solved by a small (and not serious) restriction in right recursion of the grammars, and a careful processing of the grammars by a compiler (described later). A coarse upper bound for the stack size can be determined.

The nondeterminism is a more serious problem. There is no general way to avoid it without further restriction of the grammars to the LL-parsable class. To keep the implementation simple, the actual restriction will be to LL(1). Although this may complicate the descriptions of certain behaviours, it is expected that the expressive power of LL(1) protocol grammars will not suffer much and still be sufficient, because of the ability to store context information in attributes and use them to control parsing by conditions on rules.

4.7 Relation to protocol engines

A protocol implementation is defined as a system, whose externally observable behaviour complies with the abstract specification of the protocol. A protocol engine is a hardware based protocol implementation (chapter 1 defines the term 'hardware based'). The abstract specification is often divided into smaller parts (layers). A layer implementation is a system whose actions satisfy the layer specification. The functionality of a layer is defined in terms of services it provides at its interfaces (to the higher and lower layers). These services are implemented in processing entities within each layer. The entities communicate with other entities (in the same or another layer) by exchange of service primitives over their interfaces.

A protocol grammar specifies the externally observable behaviour (I/O) of a single layer processing entity. Its implementation in a protocol pushdown automaton therefore represents such an entity. A layer engine consists of one or more protocol pushdown automata and a protocol engine consists of one or more layer engines.

To obtain a protocol grammar based specification of an entire protocol, the abstract protocol specification has to be split in separate entity specifications. These are each transformed into protocol grammars, which can then be implemented in automata. These automata have to be connected to exchange service primitives. It is not always
desirable to define and implement all functionality in grammars (such as packet buffering, assembly and disassembly) for reasons of expected efficiency, performance and memory requirements. These operations should be implemented in dedicated systems external to the pushdown automata. Because the automata have to communicate with these systems and with each other, the protocol grammars must provide a mechanism to describe that communication.

The only mechanism available to input and output information is by means of terminal symbols. The message types (service primitives) correspond to the symbols themselves, and the message parameters are stored in the attributes of the symbol, which are always sent and received with the symbol. The attribute types must match the range of values that the parameters can assume.

The transmission of a message in a certain protocol state is accomplished by putting its output terminal in the correct place of the rule leading to that state. The parameters values are defined by a set of assignment expressions, which are automatically evaluated and assigned to the corresponding attributes before the symbol and the values of its attributes are generated on an output.

A required message reception in a certain protocol state is accomplished by putting its input terminal in the correct place of the rule leading to that state. When the symbol is received, the parameter values which are also received are automatically assigned to the corresponding attributes.
Chapter 5

A Physical Implementation of a Protocol Automaton

In chapter 3 protocol grammars were formally introduced by extending the general concept of a context free grammar with attributes to store context information, arithmetic expressions over these attributes, the possibility for input and output symbols in a single grammar, and conditional rules using boolean expressions over attributes allowing context dependent behaviour. The extensions were necessary to enable the description of complex data communication protocols. Protocol grammars and the language they specify were defined formally.

In chapter 4 the general implementation model for context free grammars, the pushdown automaton, was extended with an attribute storage (tape), an attribute operator (to evaluate attribute expressions), input and output handlers and the possibility for conditional operation of the finite automaton, based on evaluation of boolean expressions by the attribute operator. This resulted in a general abstract implementation model for protocol grammars, called the protocol pushdown automaton. The language of such an automaton is defined formally. Similarly, the mapping function from protocol grammars to protocol pushdown automata which defines how to construct an automaton whose language is exactly the same as that defined by a given protocol grammar has been defined mathematically and subsequently proven.

This chapter will make a step towards the creation of an actual design for a protocol pushdown automaton for LL-1 stable protocols (which is finite and deterministic). This physically implementable design shall be called the grammar processor. It has been developed and tested in a design and simulation environment. The grammar processor forms the basis for the protocol engine architecture presented in chapter 6. Its parts form a subset of the modules in the database of the design system discussed in section 1.5 and chapter 7.
5.1 Functionality of the protocol pushdown automaton

A standard pushdown automaton contains a LIFO stack, a finite automaton and an input reader (see figure 4.1). To create the protocol pushdown automaton the following extensions were introduced (see figure 4.2):

- an attribute storage (tape) and operator to store and manipulate attributes
- an output generator
- all symbols can have (attribute) values attached
- dynamic rule enabling and disabling

The resulting automaton consists of 6 parts, whose functionality is given by:

1) **Parse stack**: push/pop values and read the top value without changing it (standard LIFO mechanism). Each value consist of a symbol, an attribute list, a list of attribute expressions and (only in case of an endmarker) another attribute list and an old environment.

2) **Attribute tape**: store the global environment and the local environments of all invoked rules. Each environment consists of a set of values corresponding to named attributes. These values can be accessed through a combination of environment and attribute name.

3) **Attribute operator**: updates current environment as symbols are processed. Operations are the allocation and deallocation of local environments (setting the current environment), evaluation of expressions over attributes and assigning new values to attributes.

4) **Finite Automaton**: a programmable finite state machine which can store a list of instructions (of the 6 types given in chapter 4) and execute them according to a fixed (grammar independent) LL-1 parsing algorithm. Uses top-of-stack value, received inputs on all channels and optional boolean value returned by attribute operator to control its actions. Generates signals to control actions of most other parts of the automaton.

5) **Input reader**: queues incoming messages (symbols with attribute values) on all input channels, and at request of finite automaton removes the frontmost symbol from a specified channel while sending its attribute values to the attribute operator.

6) **Output generator**: queues outgoing messages (symbols from the parse stack with attribute values from the attribute operator) on all output channels, and sends them over the channel as soon as possible.
The problem in designing an actual implementation of the protocol pushdown automaton is the fact that the operations are all specified in highly abstract mathematical terms (see chapter 4), for which no obvious implementation exists. Therefore these operations must first be converted into more concrete ones using correctness preserving transformations, before any real design can be made.

The construction algorithm presented in chapter 4 specifies how a protocol grammar can be mapped onto a protocol pushdown automaton for implementation: the list of instructions for the finite automaton must be created. Since a choice was made for LL-1 parsing, the algorithm uses LL-1 parse tables (see [Aho86]). These tables and the attribute expressions are encoded in some predefined way and can subsequently be stored in corresponding memories of the grammar processor. The conversion of protocol grammars to these encoded tables and expressions is done by an offline preprocessor which shall henceforth be called the grammar compiler.

5.2 Conversion of abstract operations

The operations of the protocol pushdown automaton from chapter 4 are far too abstract for direct implementation (storing lists of attribute names and expressions on the stack, storing and retrieving environments, etc.). It is necessary to convert these operations and their effects into a less abstract form before they can be physically implemented. In this section the necessary concepts will be presented.

5.2.1 Environments

An environment was defined in chapter 3 as a function that maps names (attributes) to values:

\[ \rho : A \rightarrow W \]

This can be rewritten as a chain of 2 or 3 functions (depending on the attribute):

\[\text{map}: A \rightarrow \text{Locn} \]
\[\text{ind}: \text{Locn} \rightarrow \text{Locn} \]
\[\text{mem}: \text{Locn} \rightarrow W \]  \hspace{1cm} (5.1)

such that:

\[ \forall a \in A : (\rho(a) = \text{mem} (\text{map}(a)) \lor \rho(a) = \text{mem} (\text{ind} (\text{map}(a)))) \]

The type \text{Locn} represents a location in the attribute storage (address). The function \text{map} gives the storage location where either an attribute or the location of an attribute is stored. In the latter case, the function \text{ind} returns the storage location of the attribute. Applying \text{mem} to a storage location gives the value stored in that location.
Obviously, **mem** is implementable using a standard random access memory (RAM). The translation of attribute names to locations can be done in the grammar compiler and is therefore only done once at compile time. Evaluation of **ind** has to be done during execution (at run time) but since it cannot be detected from **p** whether **ind** should be applied at all, it is left to the compiler to maintain this information and generate appropriate machine instructions for the implementation. This means that access to an attribute **a** is encoded in the instruction list either as direct access to location **map(a)** or as indirect access to the location whose address is stored at location **map(a)**.

The consequences for the grammar processor are that the attribute store of the protocol pushdown automaton which stores the environments can be replaced by a simple random access memory implementing the **mem** function. This is possible because for every attribute access the **map** function exists only in the compilation phase, and the **ind** function (when required) is actually implemented by a machine instruction and stored in the automaton (the next section shows how this is done).

### 5.2.2 Attribute expressions and action procedures

The parse stack of the abstract automaton stores grammar symbols with attribute names and evaluation expressions (and endmarkers, which are treated later). The general type for these stack symbols used in chapters 3 and 4 is \((V_T \cup V_N) \times A^k \times E^*\) with \(k \in \mathbb{N}\). In this format, \(A^k\) is a list of \(k\) names of attributes associated with the symbol. When the symbol is processed, the set of expressions \(E^*\) must be evaluated (which changes the environment), and the values of the attributes in \(A^k\) are transmitted, received or passed on (depending on the symbol type). Storing lists of attributes (names) and expressions on the parse stack is an abstract and difficult concept. Fortunately, a very good solution to this problem has been found.

One of the operations of the grammar compiler is to change all attribute names in \(A^k\) and \(E^*\) into locations (i.e. determining the **map** function). This results in a list of locations and a set of expressions over locations denoted by \(E_L\). The expressions are generally too complex to be evaluated directly in textual form. The grammar compiler must break down the evaluations in small steps which are sufficiently elementary to be implemented physically in hardware. These small steps can be encoded using an efficient binary coding scheme, called microcode. This process is called compilation. Similarly, the transmission and reception of attribute values can be represented in microcode. The resulting microcode program can be stored in a memory located in the design for the attribute operator and get a unique integer number (ID) assigned that can be used to refer to it. Since the microcode program contains
the encoded instructions for all actions on attributes to perform when a symbol is processed, it shall be called the action procedure for that symbol.

\[ V \times A^k \times E^* \]
\[ \downarrow \quad \text{(map attributes)} \]
\[ V \times \text{Locn}^k \times E_A^* \]
\[ \downarrow \quad \text{(compilation of code)} \]
\[ V \times \text{microcode} \]
\[ \downarrow \quad \text{(store code, assign ID)} \]
\[ V \times N \]

The parse stack of the grammar processor therefore only has to store the symbol and the unique identification number of the corresponding action procedure (a conversion for endmarkers will be given later). The microcoded expressions as well as the code to send or receive the attribute values of the symbol are stored in a program memory located in the design of the attribute operator. The mapping from action procedure number to an entry point in the program memory is done through a simple address lookup table. This can easily be implemented using standard techniques.

The attribute operator is now described by a function atop:

\[
\text{atop: } N \rightarrow (\text{mem} \times W^* \rightarrow \text{mem} \times W^* \times B[8,11])
\]

Given an action procedure number, atop takes a memory (function mem) and an optional (input) list of values, and transforms it into another mem (by changing its contents) and generates an optional (output) list of values and an optional boolean output (rule enable). The remainder of section 5.2 will be used to obtain a specification for atop.

5.2.3 Nonterminal symbol processing

The expansion of a nonterminal involves the creation of a new environment. The old environment must be remembered and restored at the end of the expansion sequence. Environments are stored on the attribute tape of the protocol pushdown automaton which can be implemented as a standard RAM. An interesting observation from chapter 3 is that every environment is partitioned into a global and a local part, and that only the local parts of environments have to be remembered, created, deleted and restored.
The global environment is always accessible (from every rule) and contains all global attributes. Since global attributes cannot be created dynamically, the size of the global part is constant for a certain grammar. The global attributes can hence be stored in fixed locations in the attribute RAM (determined by the compiler), which can be represented by absolute addresses.

5.2.3.1 Attribute passing mechanism

The local part is newly created whenever a nonterminal is expanded, and deleted when the expansion sequence is completely processed (indicated by endmarkers). It contains the local attributes needed during processing of the entire sequence. The nonterminal action procedure is used to pass attributes back and forth between the invoking and invoked rules. According to the abstract definition, this requires copying of attribute values from the original environment to the new local environment where they can be changed during symbol processing, and finally copying them back to the restored original environment.

The idea for the physical implementation of this mechanism is to pass references (i.e. pointers) to these attributes rather than their values. This saves the time needed for copying the values of the attributes twice, but requires an additional memory cycle for every access to the attribute value itself (this is where the function \texttt{ind} is used). A local environment then contains values for all local attributes for a rule, except those that were actually passed as left hand side (LHS) attributes. For these, it instead contains references to the locations were the attribute values are stored. This method is expected to be generally more efficient, since it does not require copying data. It is possible because evaluations are not allowed to have side effects, and because it is not allowed to assign values to LHS inherited attributes.

Since local environments are created dynamically, their location is not known in advance and the locations of local attributes can therefore not be precomputed by the compiler. This is solved by using a pair (base, offset) as a location. The base element contains the start position of the local environment and offset is the relative position of an attribute with respect to base. By storing the local part in a contiguous part of the attribute memory, the offset values are fixed for every rule and can be precomputed by the compiler. The base value can be computed at run time and maintained in an internal register in the attribute operator implementation.

Formally: \( \text{Locn} \in \mathbb{N} \cup \mathbb{N} \times \mathbb{N} \cup \{ \text{unallocated} \} \)

Expanding \(<n, \alpha_1, e_1>\) in \(\rho\) using rule \(<c, <n, \alpha_2, e_2>, \eta>\) results in \(\rho''\) where \(\rho' = \text{EV}(e_i)(\rho)\) and \(\rho''' = \rho'[A_i / \bot][\Theta(\eta) / T]\) and \(\rho'' = \rho'''[\alpha_2 / \rho'(\alpha_i)]\)
Using the above method and equation 5.1, the compiler generates code such that:

\[ L'(\alpha_i) = \text{ind}(L''(\alpha_2)) \]

where \( \rho' = m' \circ L' \) and \( \rho'' = m'' \circ L'' \)  

(5.2)

Note that \( L', L'' \) and \( L''' \) are determined and computed at compile time and as such only 'exist' in the compiler. The physical implementation only contains \( m', m'' \) and \( m''' \). Because the environment transformation from \( \rho' \) via \( \rho''' \) to \( \rho'' \) does not change the values of attributes \( \alpha_i \) in \( \rho' \), the combination with equation 5.2 results in:

\[ m' \circ L'(\alpha_i) = m'' \circ L'(\alpha_i) = m''' \circ \text{ind} \circ L''(\alpha_2) \]

The compiler will automatically insert code to evaluate the \( \text{ind} \) function whenever an attribute from \( \alpha_2 \) is needed. Therefore \( \rho'(\alpha_i) = \rho''(\alpha_2) \) and the substitution part of the transformation can now be implemented by setting up the \( \text{ind} \) function for these attributes (by allocating \( |\alpha_2| \) locations and storing locations of attributes \( \alpha_i \) at those positions). Then it follows directly that \( \rho'(\alpha_i) = \rho''(\alpha_2) \).

5.2.3.2 Local environment allocation

The remaining environment allocation operation which is part of a nonterminal expansion \( \rho''' := \rho'[A_L / \bot][\Theta(\eta) / T] \) should be interpreted as replacing the current local environment by a new one, where all additional local attributes of the right half side of the expansion rule are allocated, but with undefined value. An obvious implementation is to allocate a contiguous space of attribute RAM large enough to hold the values of all these attributes and to set it as the current environment.

The allocation and deallocation of local environments can be done on a LIFO basis (stack segments). The local environments of all rules in the invocation list form a stack with the local environment of the current rule on top. A register (called base) is used to hold the location of the top element of the environment stack (base element for Locn pair). To allocate a new local environment of size \( k \) on the environment stack, the value in the register is incremented by \( k \). To deallocate that environment, the value in the register is decremented by \( k \).

So far the complete list of operations to be performed by the attribute operator for nonterminal expansion has become:

- evaluate \( \epsilon_i \) to change \( \rho \) into \( \rho' \)
- allocate space for storage of \( |\alpha_2| \) locations and all local attributes not in \( \alpha_2 \).
  This results in temporary environment \( \rho''' = m''' \circ L''' \)
- store \( L'(\alpha_i) \) at \( m'''(L''(\alpha_2)) \), resulting in \( \rho'' \).
5.2.3.3 Enable conditions and rule selection

From the previous section, it would seem that the environment operations for expansion could all be performed by a single action procedure whose identification number is pushed on the parse stack together with the nonterminal symbol. However, nonterminal expansion also requires finding a rule and testing its enable conditions. The enable conditions must hold in \( p' \) and may depend on LHS inherited attributes (i.e., on \( p'(\alpha_1) \)). In the grammar these conditions will be written in terms of attributes from \( \alpha_2 \) whose names can be different for each rule. However, after application of \( L \) by the compiler the names of the attributes are no longer relevant and all conditions can be expressed in terms of the storage locations \( L'(\alpha_1) \) which can be found in \( m'' \) at locations \( L''(\alpha_2) \). Because of this, the enable conditions can be evaluated in the temporary environment \( p'' \). The operations for expansion are divided over action procedures as follows:

- The action procedure for the nonterminal itself evaluates \( \varepsilon_1 \), then allocates space for \( |\alpha_1| \) locations (resulting in a temporary environment \( \rho_T \)) and finally stores \( L'(\alpha_1) \) at \( m_T(L_\tau(\alpha_2)) \). See figure 5.6 on page 111. The identification number for this procedure is stored on the parse stack with the nonterminal.
- Every rule has two corresponding action procedures, one for evaluation of its enable condition and one for allocation of the remaining RHS attributes. These are stored in the parse tables of the finite automaton, together with the rules themselves.
- For every applicable rule, the finite automaton emits the enable condition action procedure number and tells the attribute operator to execute the corresponding microprogram (returning a boolean value). If the result is true, the rule is selected to expand the nonterminal on the parse stack, otherwise the next applicable rule is tried (this implements the abstract test instruction).
- When a rule is selected for expansion, the finite automaton emits the attribute allocation action procedure number for that rule and tells the attribute operator to execute it. This procedure will allocate space to hold all the RHS local attributes which were not in \( \alpha_2 \). This environment is concatenated to \( \rho_T \) to obtain \( \rho'' \).

Note that the size of a local environment is fixed (precomputable by the compiler) for each rule and can be hard-coded into the allocation and deallocation microcode. The main conclusion for the grammar processor is that all environment allocations and deallocations and all attribute expressions are implemented using microcode programs, the so called action procedures, which can be linked to the corresponding symbols and rules by means of a single unique identification number.
5.2.4 Environment restores for endmarkers

Every expansion sequence ends with an endmarker on the parse stack, with the exception of endless loop constructs (see section 4.6.2). The abstract operations for this special symbol are the attribute evaluations of the LHS synthesized attributes, deallocation of the local environment and restoration of the local environment of the previous rule in the invocation list. In the abstract protocol pushdown automaton all information required for this operation is stored in an endmarker symbol on the parse stack. The abstract operation seems difficult to implement, but using the ideas presented earlier in this section it can be converted into a simple one.

For an endmarker \(<\alpha, \rho'(\alpha), \alpha_1, e, \alpha_2>\) encountered on the parse stack in an environment \(\rho\) the operation results in a new environment \(\rho''\) and sets it as the current one, where:

\[
\rho'' := (\rho'(\alpha) \cup \rho''(\alpha))|\alpha_1 / \rho''(\alpha_2)\] and \(\rho''' := \text{EV}(e)(\rho)\)

By construction of expansion, it follows that \(\text{ind}(L(\alpha_2)) = L'(\alpha_1)\) and since for any global attribute \(a: L_1(a) = L_2(a)\) for all \(L_1, L_2 \in \text{map}\), and because evaluating \(e\) can never change locations of attributes, it follows that \(\rho'(\alpha_1) = \rho''(\alpha_2)\) and therefore

\[
(\rho'(\alpha) \cup \rho''(\alpha))|\alpha_1 / \rho''(\alpha_2) = \rho''(\alpha_2)
\]

which means that the substitution is superfluous and can be omitted entirely. The remaining operations are:

\[
\rho''' := \text{EV}(e)(\rho)\] and \(\rho'(\alpha) := \rho'(\alpha)\)

The evaluation is microcoded by the compiler in an action procedure. The local environment deallocation and restore are done by simply decrementing the value of the base register by the size of the environment \(\rho'(\alpha)\) to be deallocated. The microcode instructions to do this are encoded and appended to the action procedure by the compiler.

The consequence for the grammar processor is that since the attribute lists \(\alpha_1\) and \(\alpha_2\) are no longer required during restore operations and the old environment \(\rho'(\alpha)\) can be retrieved without storing it on the stack (as in the abstract operation definition), these elements do not have to be stored on the parse stack. Only the action procedure to evaluate \(e\) and adjust base is needed along with the endmarker symbol itself. In those cases where no endmarker may be used, its microcode instructions are not put in a separate action procedure but appended to the action procedure of the last symbol of the rule. Endmarkers are only omitted in places where this preserves the correctness of the implementation.
5.2.5 Attribute access, allocation and deallocation

In chapter 4 it was shown that all existing (allocated) local environments form a stack. Hence allocation and deallocation of attribute memory can be done on a LIFO stack basis. The elements on this stack are the local environments and a single global environment (at the bottom since it was created when the automaton started working). The mechanism for (de-)allocation becomes rather simple.

For the physical implementation, suppose that a new environment is placed above the previous one in attribute memory (i.e. top of stack address increments when an element is pushed) and let base be its address. Thus all addresses above or equal to base are not allocated to any attribute in any invoked rule, and all addresses below base are allocated to some attribute in some invoked rule. Let the amount of space required to store the attributes of a rule r be denoted by size(r). This value can be computed by the compiler. It is the same for all invocations of r. The value of base is maintained in a base register in the design for the attribute operator and it equals the highest address of the current local environment plus one. Changing base automatically changes the current local environment because location offsets used for all local attributes now refer to other memory locations (and thus to other attributes).

The conclusion is that the attribute operator must contain a register whose value always equals the top address of the current local environment in the attribute RAM. This register is called base. Local attributes are always accessed as an offset from this base value. Passed attributes will require an additional read access (to get the real attribute address). Global attributes will be located in a set of fixed locations at the bottom of the attribute RAM and can be accessed using absolute addresses. Allocation of a new local environment is done by adding the size of the new environment to the current value of base. Deallocation is done by subtracting the same value. The attribute operator is programmed to perform the necessary actions by means of action procedures which it must execute at command of the finite automaton.

5.2.6 Specification of the attribute operator

If $e \in E^*$ is an expression list, then $e_L \in E_L^*$ represents the same expression list after all attribute names have been transformed (by the compiler) into locations. The semantic evaluation EV is transformed into a mem transforming function EV$_L$ by the compiler:

$\text{EV} : E^* \rightarrow \text{mem} \rightarrow \text{mem}$

defined such that:

$\text{EV}(e) \underbrace{\circ L} = (\text{EV}_L(e_L) \underbrace{\circ m}) \circ L$
Similarly, CV is transformed into:

\[ CV_L : E_L \rightarrow \text{mem} \rightarrow \mathbb{B} \]

such that:

\[ CV(c) (m \circ L) = CV_L(c_L) (m) ; c \in BE \]

Let \( \rho = m \circ L \) be the current environment (\( m \in \text{mem} \) and \( L \in \text{map} \)) and let \( k \in \mathbb{N} \) be the identification number for the action procedure of the symbol on top of the parse stack. Furthermore, let \( m', m'', m''' \in \text{mem} \) and \( w \in W^* \). If absence of values (empty lists) is denoted by \( \emptyset \), then the generated microcode must be such that the Attribute Evaluator behaviour is described by the following definition of atop:

For an input terminal \( \langle t, \alpha, \varepsilon \rangle \):

\[
\text{atop}(k, (m, w)) = (m', \emptyset, \emptyset) ; m' = EV_L(\varepsilon_L) (m[L(\alpha)/w])
\]

For an output terminal \( \langle \alpha, \alpha, \varepsilon \rangle \):

\[
\text{atop}(k, (m, \emptyset)) = (m', w, \emptyset) ; m' = EV_L(\varepsilon_L) (m) \text{ and } w = m'(L(\alpha))
\]

For a nonterminal \( \langle n, \alpha, \varepsilon \rangle \):

\[
\text{atop}(k, (m, \emptyset)) = (m', \emptyset, \emptyset) ; m'' = EV_L(\varepsilon_L) (m)
\]

\[ m' : (\forall l \in \text{Locn}: (l < \text{base} \Leftrightarrow m'(l) = m''(l))) \land
m'(\text{base} \ldots \text{base} + |\alpha| - 1) = L(\alpha) \]

For every rule \( r = \langle c, \langle n, \alpha, \varepsilon \rangle, \eta \rangle \) allocation procedure after selection:

\[
\text{atop}(k, (m, \emptyset)) = (m', \emptyset, \emptyset) ; \text{base'} = \text{base} + \text{size}(r), \text{changes environment to } \rho'
\]

For an endmarker \( \langle x, \rho', \alpha, \varepsilon, \alpha_x \rangle \) concatenated to a rule \( r \):

\[
\text{atop}(k, (m, \emptyset)) = (m'', \emptyset, \emptyset) ; \text{base''} = \text{base} - \text{size}(r), \text{changes environment to } \rho''
\]

For every rule \( \langle c, \langle n, \alpha, \varepsilon \rangle, \eta \rangle \) condition procedure:

\[
\text{atop}(k, (m, \emptyset)) = (m, \emptyset, b) ; b = CV_L(c_L) (m)
\]

It is the task of the compiler to perform the transformation of global attributes to fixed locations and of local attributes to (base, offset) locations, to break down the resulting expressions into small steps that are each directly executable by hardware, and to generate code to allocate and deallocate memory by means of adjusting the base register. The design of the hardware architecture and the code generator part of the compiler were done at the same time, because a change in one of the two has a great impact on the other.
5.3 The grammar processor

The proposed architecture is shown in figure 5.1. It contains a fairly complex finite state machine called the Pushdown Controller which globally controls all actions taken by the processor (it implements the finite automaton part of the protocol pushdown automaton).

![Diagram of the grammar processor]

**Figure 5.1** The architecture for the grammar processor.

The LIFO memory named Parse Stack will store unprocessed parts of invoked production rules (as with normal PDAs), consisting of symbols and action procedure numbers. The two blocks called Output Writer and Input Reader are interfaces that allow asynchronous input symbol arrivals from a channel and output symbol generations onto a channel by serving and buffering data for a number of channels. They communicate with the Pushdown Controller by means of a handshake mechanism and they merge attribute values with symbols before generation, resp. extract attribute values from symbols after receipt. The channel control lines (ch#) deter-
mine on which input or output channel a terminal should be sent or received, according to the terminal partition to which that terminal belongs (defined in the protocol grammar). Finally, there is a rather complex block called Attribute Evaluator, whose function is to maintain all attributes for the grammar. It implements both the attribute operator and the attribute RAM. Hence, it must be able to perform any desired operation that is allowed by the compiler for the protocol grammar interface language (i.e. evaluate any allowed expression) and maintain attribute memory.

5.4 The parse stack

Following the implementation concepts outlined in section 5.2, the parse stack will only store symbols and corresponding action procedure numbers. Every word stored on the stack thus contains two fields, one representing the symbol and one for the procedure. Since there is at most one action procedure for every symbol in every rule, one for the endmarker of each rule, one for the enable condition and one for the allocation of each rule, and the number of rules and their length is finite, the number of action procedures is also finite. The sizes of the two fields of the parse stack are therefore finite and can be extracted from the grammar.

The functionality of the stack can be taken from the formal definition (chapter 4):

- push: Stack x Elem → Stack
- pop: Stack → Stack x Elem
- rd.to.s: Stack → Stack x Elem
- reset: Stack → Stack (for initialization)

where:  
Elem = Symbol x Integer  
Integer = {0, ..., n} for some fixed n ∈ N  
Stack = Elem*

The operation is completely characterized by the following axiomatic definition:

\[
\begin{align*}
\text{pop}(\text{push}(s, e)) &= (s, e) \\
\text{rd.to.s}(\text{push}(s, e)) &= (\text{push}(s, e), e) \\
\text{reset}(s) &= \emptyset
\end{align*}
\]
The operations pop and rdtos only apply to non-empty stacks, otherwise the result is undefined. The operation push only applies to stacks that are not completely full. This limitation is the result of the finiteness of the implemented stack memory. Figure 5.2 shows a standard finite stack implementation.

5.5 Invocation of action procedures

This section explains how and when the Pushdown Controller tells the Attribute Evaluator to perform a required environment update when a symbol is parsed.

The concept of action procedures was invented to enable an easy implementation of environment manipulations whenever a symbol is parsed. Instead of whole lists of attributes and expressions, a single integer number representing a microprogram stored in the Attribute Evaluator should now be stored on the parse stack. When executed, this microprogram would result in precisely those changes in the environment that are necessary for the symbol being processed or the rule being tested or used for expansion. Clearly the Attribute Evaluator must be able to execute the microprograms, which means that it is basically a custom microprocessor whose architecture is especially adapted to handle attribute management. The instruction set must be powerful enough to express all required operations. For a very detailed report on the interface language, the architecture of the Attribute Evaluator, the microcode instruction set and the method used to generate the microcode programs from high level expressions, see [Bloks93a] and [Bloks93b].
The operation sequence is as follows: when a symbol is parsed (popped of the parse stack), its action procedure number is also popped from the stack and sent to the Attribute Evaluator (see figure 5.1). The Pushdown Controller activates the execution of this procedure by sending a call_proc signal, and then waits until the execution is complete, as indicated by the done signal returned from the Attribute Evaluator. The action procedure always has to complete before another procedure is called, since results of the first action procedure might be needed in the second.

While an action procedure for a symbol is being executed, other symbols can still be parsed by the Pushdown Controller simultaneously, as long as they do not require the invocation of an action procedure. Not all symbols will have attributes and expressions to evaluate. Such symbols will not need any action procedure. To avoid the unnecessary overhead of invoking an empty action procedure, a special action procedure identification number has been defined (reserved) to indicate this.

5.6 Enable conditions for production rules

This section explains how the Pushdown Controller implements enable procedures.

Every rule can have an enable condition (a computable boolean expression). When the expression evaluates to true, the production rule is enabled, otherwise it is disabled. If no expression is explicitly defined the rule is enabled by default. Computability implies that the condition expressions may depend on global attributes, the LHS inherited attributes (which have already been evaluated when enable conditions are being computed) and on time in particular. The time dependence means that the expression result can change without any observable or externally triggered change in the state of the system. It is not possible to capture time dependence in normal (static) attributes.

Whenever the symbol on top of the parse stack is a nonterminal, all applicable production rules for that nonterminal must be examined cyclically, to see if their enable conditions are true. The first enabled applicable rule found in this way is then used to expand the nonterminal. The mechanism for evaluating these boolean expressions has already been explained in section 5.2. The expression is compiled into an action procedure which will return the boolean result value to the Pushdown Controller (signal enable in figure 5.1). The action procedure numbers for the condition expressions of all rules are stored in the Pushdown Controller.
5.7 Input and output operations

For output generation, an output terminal must be combined with a list of values of its attributes. The symbol is the same as the one taken from the stack (output terminal), and the attribute values are generated by the Attribute Evaluator under control of an action procedure. Its execution results in a stream of words on the ext_out output, whose concatenation is an exact replica of the data structure maintained by the Attribute Evaluator for the corresponding attributes of the output terminal.

The Output Writer combines symbols and their attribute values. It contains a lookup table which stores the number of attribute words for each output terminal. When a terminal is received from the Stack and Pushdown Controller, the Output Writer determines the number of data words to expect from the Attribute Evaluator, then accepts that number of words using a one-way handshake mechanism. The Output Writer is then ready to accept another symbol. The symbol and all data words are buffered. Eventually, the symbol and data words will be transmitted on the symbol output.

Input acceptance is a similar process. The Input Reader accepts and buffers incoming messages. The first word of every message must be an input terminal. From a lookup table, the Input Reader determines how many data words must follow. The set of all current input terminals is sent to the Pushdown Controller for processing, and the data words are stored to be transferred to the Attribute Evaluator when an input is accepted. The Pushdown Controller tries to process all available input terminals sequentially. In case of erroneous input this may not be possible. Instead of halting operation, the input can be discarded. This function has also been assigned to the Input Reader, and can be specified for each input terminal separately in the protocol grammar definition. At command of the Pushdown Controller, the Input Reader must discard an input terminal with all its data words (signal discard of the Input Reader in figure 5.1).

5.8 The Pushdown Controller

The finite automaton part of the abstract protocol pushdown automaton is implemented in the Pushdown Controller. To enable deterministic parsing, it uses an LL(1) parsing algorithm, which requires storage of a number of data structures which can be directly computed from the grammar. It must be able to read symbols from the stack, process them, generate outputs, accept inputs and control the Attribute Evaluator for environment transformations and enable condition compu-
A Physical Implementation of a Protocol Automaton

An overview is given in figure 5.3. A more detailed report on the initial design and operation of the Pushdown Controller can be found in [Jacobs91].

The Finite State Pushdown Controller is a standard finite state machine which controls all actions. It handles the I/O handshaking, controls the Attribute Evaluator and the parse stack. Its states represent the states of a general LL(1) parser for protocol grammars.

The Symbol & Action Decoder will compare every input symbol to a terminal on top of the stack. When the symbol on the top of the stack is an input terminal, the next input symbol must match it exactly. Otherwise, the input string cannot be parsed (this is usually the result of a protocol procedure error, where the remote node did not act according to the rules of the protocol) and the input must be discarded.

---

**Figure 5.3** Internal architecture of the Pushdown Controller.
The decoder will also examine the symbol on top of the parse stack itself, and indicate the type of this symbol (nonterminal, input or output terminal, or endmarker) to the Finite State Pushdown Controller. This distinction is necessary, because the required actions for parsing each type of symbol are completely different. The decoder also generates the channel number on which any input is expected or on which an output must be sent (used by Output Writer and Input Reader), and it detects the reserved action procedure number, used to indicate that no real action procedure needs to be called.

The Rules-ROM contains the expansion (symbol-) sequences as defined by the production rules with the corresponding action procedure numbers for the Attribute Evaluator (the sequences are in reversed order, because the last data written will be the first popped). It is needed when a nonterminal on top of the stack must be replaced by the RHS of one of its production rules. An address counter is set to the start address of the requested production rule data in the Rules-ROM and all symbols of that rule are copied from the Rules-ROM onto the stack. The Pushdown State Controller can detect the end of the rule during copying by means of the end bit in the Rules-ROM. This bit will be set at every address whose contents are the last to push on the stack during expansion (i.e. the leftmost RHS symbols).

The Next Rule Selector contains a state machine and a table with look ahead sets, Rules-ROM start addresses and enable condition and allocation action procedure numbers for all production rules of the grammar. The state machine is activated when a nonterminal is detected on the top of the parse stack. It will then test all rules for that nonterminal until one is found that can be used to expand it. If there are at least two rules for the given nonterminal, of which at least one has a look ahead set containing input terminals, then the LL(1) parsing method requires that the next input symbol must be an element of that rules' look ahead (LA) set. This condition is indicated by a flag (match-LA) which is stored in the LA table and computed by the grammar compiler. Testing an applicable rule is done according to the following method:

- If look ahead matching is required for the rule (match-LA), the next input symbol must be an element of the look ahead set. If an input symbol is available on every channel from which the rule can accept inputs, but none of them is in the look ahead set, the rule is no longer considered applicable in the current state. If an input symbol is available which is an element of the look ahead set, the rule's enable condition is evaluated and if the result is true, the rule is used for expansion. If the result is false or if there was no input symbol
available at all, the rule remains applicable but no other action is taken at the moment. Testing continues with the next applicable rule.

- If LA matching is not required for this rule, no input symbol is needed to continue. In this case the enable condition is evaluated, and if the result is true the rule will be used for expansion, otherwise the current rule remains applicable and the next one is tested.

To enhance performance, the Next Rule Selector will never recompute an enable condition that does not depend on time after it last started looking for a rule. If no condition ever becomes true, the processor cannot continue. This deadlock situation is detected by the Next Rule Selector and requires special handling by the Finite State Pushdown Controller. It is usually the result of an error in the grammar specification (incomplete specification).

Note that the underlying grammar does not necessarily have to be LL(1). If the enable attributes resolve any further conflicts, the machine is still deterministic. This allows slightly more powerful grammars and thus more complex languages than conventional LL(1) parsers. If the enable conditions do not resolve conflicts, then it is impossible to predict which rule will be chosen from a set of enabled rules. The result can be regarded as a nondeterministic choice from this set. This normally is an undesired property. However, the grammar compiler does not detect this situation.

5.9 The Attribute Evaluator

5.9.1 The global architecture

The most complex part of the grammar processor is the Attribute Evaluator, whose main function is to store and to manipulate the attributes defined in the protocol grammar. It must be able to execute any requested action procedure and return a boolean flag as a result code. To evaluate expressions it should at least contain an ALU, capable of executing the operations used in the grammar. Its global overview is shown in figure 5.4. To store and retrieve attributes efficiently, a very fast dual access RAM is used. Address generators provide the access addresses for the RAM on a register direct with displacement base (in local environments) or by absolute addressing (in the global environment). A timer unit continuously keeps track of the system time, which is needed for time-out constructs (located inside the Arithmetic Unit). A micro controller (μ-CTL) generates all necessary control and data signals to coordinate all operations within the Attribute Evaluator.
The Attribute Evaluator has been designed as a multistage pipelined processor unit to increase performance. A high performance of this unit is extremely important for obtaining very high throughput protocol engines. It was designed for efficient implementation and execution of action procedures. A more detailed report on the initial design and operation of the Attribute Evaluator can be found in [Lunteren91]. See also [Bloks93b].

![Diagram of the Attribute Evaluator](image)

**Figure 5.4 The Attribute Evaluator.**

The micro controller (μ-CTL) contains a ROM in which all action procedures are stored in the form of microcode. A lookup table translates a procedure identification number into a start address in this ROM. The controller starts executing instructions from that address, while generating control and data vectors for the other parts of the Attribute Evaluator.

5.9.2 Instruction execution and pipelining

Instruction execution takes 5 (pipeline) cycles:

- **cycle 1:** μ-CTL generates control vectors and computes next instruction address.
- **cycle 2:** Address generators compute addresses for RAM access and update base address registers (if necessary).
cycle 3: RAM and/or external input are accessed for source operand (if necessary).
cycle 4: ALU processes data and computes a result (fast operations).
cycle 5: Result is written to destination.

As long as no conflicts occur in the pipeline, it remains full. This means that one instruction is completed every cycle. Conflicts may hold up or clear (part of) the pipeline structure, thereby decreasing performance, but all conflicts are automatically resolved entirely in hardware (i.e. the pipelining effect is transparent to software running on the processor).

All ALU operations may take one or more cycles. For example an ADD will probably take one cycle, but a multiply or bit shift over multiple bits will take longer unless the ALU contains special hardware for these operations. The arithmetic unit can inform the µ-CTL that the current operation will not complete at the next clock by means of the busy lines. The pipeline will then be put on hold until the ALU has finished its operation.

5.9.3 Data types and operations

The attribute data types are defined in the protocol grammar. The description language and compiler allow a small set of predefined attribute types, but also the definition of arrays, structures, ranges and enumerated types (see [Bloks93a]). The code generator part of the compiler must handle all these attribute types. More important for fast operation is that the Attribute Evaluator must be able to handle these attribute types efficiently. Therefore, special care must be taken in the design of the Attribute Evaluator to make sure that all allowed attribute types and operations can be implemented efficiently.

For a complete list of all attribute data types and operations, see chapter 7. For an efficient implementation of operations on attributes, some requirements for ALU operations and the address generator can be derived:

- Operations on simple types, such as 'words' should be very fast and take only 1 micro instruction. This requires the ability to denote 2 sources and 1 destination within one instruction (example: \( A := B + C \))
- To enable copying of large data blocks (structure and array types) in memory, it is necessary to have either a special data block copy operator or a register direct with displacement and post/pre-increment addressing mode. This last option was chosen, because register direct with displacement addressing is also required for accessing array elements and LHS attributes (passed as
pointers).

- For packing and unpacking of packed operands, bit shift operations are required to shift operands right or left by a variable number of bits. For the same reason, sign extend operations from a variably located sign bit are needed.
- Special operations, such as value wrapping and bounds checking are needed for assignments to some types of attributes and array indexing.
- For all standard arithmetic and logic operations in the grammar interface language, there should be a similar ALU operation.

A detailed formal description of the mechanisms used for the generation of nearly optimal microcode from high level expressions can be found in [Bloks93b].

### 5.10 Memory usage of local environments

Whenever a production rule $p$ is invoked to expand a nonterminal $N$, space must be allocated in the attribute memory. In section 5.2, it was shown that this space is used to store:

- The locations (addresses) of the LHS inherited attributes of $N$.
- The locations (addresses) of the LHS synthesized attributes of $N$.
- All synthesized local attributes on the RHS of $p$.
- All inherited local attributes on the RHS of $p$.

The evaluation of expressions $EV$ and $CV$ has been broken down in many small steps. This usually requires the storage of intermediate values, for which memory should also be allocated. The sizes of all these attributes can be determined during compilation because attributes have to be declared before they can be used, and production rules have a finite length (i.e. both the size and number of attributes for each rule will be known and finite in the design system). The total memory allocated for the invocation of a rule will be called its workspace.

The maximum total amount of storage needed at any given point during the grammar execution is much harder to compute. At any time, memory consumption depends on the invocation list and the required amount of storage for every rule. Finding the maximum requires computation of all possible invocation lists and their total memory consumption, and taking the maximum of those values. The size of the memory allocated for global attributes must be added. Symbolic execution is most likely to provide a good tool for determining all possible invocation lists.
5.11 Attribute allocation map

The attribute memory is needed for storage of:

- System variables, such as the start of the current workspace and the amount of available free memory (for out-of-memory detection). The number of variables stored here is fixed (independent of the grammar). This system environment is always present and allocated when the system starts up.
- Global attributes. The size is determined at compile time, and allocation is done when the system is started.
- Local synthesized attributes. The size depends on production rule and is computed at compile time for each rule. Allocation is dynamic and takes place when a new rule is invoked.
- Local inherited attributes. Allocation is dynamic at the start of a rule.
- Addresses of LHS attributes. Whenever a nonterminal is expanded that has attributes, the addresses of these attributes must be stored in the workspace of the new rule. Allocation is dynamic during expansion.
- Temporary variables, needed for expression evaluation. The amount of storage for this equals the maximum amount to evaluate any single expression for the new rule, since only one expression is being evaluated at any time. Size can be computed at compile time and differs for each production rule. Allocation should again be dynamic when a new rule is started.

At any time, the total memory space will be divided into a number of allocated segments:

- System Segment (system environment)
- Global Attributes Segment (global environment)
- n Local Attribute Segments (local environments or workspaces)
- Free Space Segment (unallocated memory)

The size of all global attributes, and their addresses are determined at compile time. The global attributes segment is located just above the system segment. The first free location above this segment is the address of the first workspace. Workspaces are always allocated at the lowest free location in RAM. The address of the current workspace is always stored in the standard base registers in the address generators, and a shadow copy will be kept in the system segment in case it is needed (the base registers are write-only in the current design). When a new segment must be allo-
cated, the standard base registers and the shadow variable must be adjusted. The placement of attributes within a workspace will be as follows:

- **low** – addresses of LHS inherited attributes
- - addresses of LHS synthesized attributes
- - RHS inherited attributes
- **high** – RHS synthesized attributes

Figure 5.5 shows a memory map of the attribute memory. Temporary variables are missing from this map, because they will be put in free space directly above the current workspace. It is assumed that there is enough free space available to store those temporary variables, but no check is performed to guarantee that this assumption is actually valid.

![Figure 5.5 Memory map of the attribute RAM.](image)

The workspace allocation mechanism is explained using figure 5.6. The left half shows a workspace and the value of the base registers pointing just above it. During symbol parsing, the local attributes and pointers to passed attributes can be accessed using negative offsets from the base register and the temporary variables are stored from offset 0 and up into free space. For every rule, the exact size of its workspace (wsSize) and the number of LHS attributes (#lhs attr.) is fixed and computed by the
compiler and this information is used by the code generator to determine at compile time if a value located at a specified (negative) offset is just a pointer or an attribute.

When a nonterminal is detected on top of the stack, its inherited attributes are computed after which pointers to all of its attributes are written to offsets 0 and up into free space (at that moment no temporary variables are in use, since these are used only during expression evaluation). Then the Pushdown Controller starts looking for a new rule, and may call enable procedures. Enable procedures will therefore operate in an environment as shown in the right half of figure 5.6. Pointers to LHS attributes of the potential new rule can be found just below offset 0 (these attributes may be used by enable procedures), local attributes of the current rule are by definition inaccessible and temporary variables may be stored above the last attribute pointer (at offset 0). As soon as the Pushdown Controller has found a suitable rule, space for its local attributes is allocated by adding the total size of these attributes and the number of passed attributes to the base value. At this moment, the situation is the same as in the left half of figure 5.6, except that base points to the top of the new workspace. When a rule finishes, memory deallocation is simple done by subtracting the workspace size for that rule (wsSize, a compiler computed constant for each rule) from the base value.

![Diagram of Memory Allocation for new rules.](image-url)
5.12 Conclusion

This chapter has presented a possible design for the actual implementation of a protocol pushdown automaton. The difficult and abstract operations by which the original automaton was specified have been converted to less abstract ones, for which implementations could be found. This required the introduction of some design choices, such as action procedures and the use of pointers to passed attributes. The high level architecture of the resulting grammar processor closely resembles that of the original protocol pushdown automaton. The operations of the basic blocks in the high level diagram have been explained in some detail.

Using this architecture, each stable LL-1 protocol grammar can now be implemented in a finite grammar processor. Therefore, the means are now available to construct an entire protocol engine with multiple grammar processors from a set of protocol grammars describing a communication protocol.
Chapter 6

Protocol Engines

In the previous chapters an implementation model for protocol layer entities was presented, together with a formal description technique and a method that will enable the automatic generation of such an implementation. Not all protocol functionality is included in this model, however. One of the most important aspects in protocol implementation is the management of packet buffers, and the assembly and disassembly of packets. This chapter will present a proposal for a general protocol engine architecture, where these operations are performed by dedicated hardware to obtain extremely high performance. It shall also be shown how engines containing multiple grammar processors can be constructed, and how these grammar processors can communicate with each other and with the packet management system.

6.1 Packet management system

In every layer of a protocol stack, packets are generated, received and analysed. These operations can be described in a protocol grammar, and hence implemented in the grammar processor, if packets are maintained as attributes and special attribute operations for these manipulations are included in the grammar. Such a solution would have a number of drawbacks, however.

- Packets can have any (finite) size, which would require dynamically sized attributes. This is difficult to implement and not included in the memory management system of the grammar processor.

- Packets would have to be stored in attribute memory, which therefore has to be very large and still be accessible very fast.

- All packets would always have to pass through grammar processor(s), even though their contents are usually not needed there. This has a major negative
impact on achievable performance. It would be far better if each processor only received the data it actually needs.

- The retransmission feature of most protocols requires that transmitted packets are stored until they are acknowledged, and retransmitted if acknowledgment takes too long or if asked for by the receiver. This storage requirement does not fit in the attribute storage mechanism, causing the packet to be rebuilt from scratch every time it had to be retransmitted. The data field would have to be generated by a higher layer grammar processor where exactly the same problem pops up. In the worst case, all grammar processors on the transmitter side would be involved in the retransmission of every single packet. This drastically reduces performance in such cases.

The inevitable conclusion is that packets should not be maintained as attributes in the grammar processor, but rather in a separate packet management system. All operations regarding packets are performed in this system under explicit control of the grammar processors. The packet management system is just another standard library module for the protocol engine.

6.1.1 Packet storage and transfer

A grammar processor that manipulates packets must be able to send commands to the packet management system and get results back. When it has completed operations on a packet, that packet can be transferred to another grammar processor for further processing. It depends on the structure of the underlying memory system how this is done. The two inherently different strategies for packet storage and transfer are:

1) Local packet memories for every grammar processor. For transfer, data has to be moved to another physical memory. Packets can be stored in FIFOs where they are sequentially accessed and processed by a packet (dis-) assembler, which adds (extracts) relevant information. The rest is passed on to the FIFO for the next grammar processor.

2) Shared packet memory for all grammar processors. Packets are not physically moved at all. Instead, references to data (pointers) are passed between grammar processors. All operations are performed using pointer manipulations.

Each method has advantages and disadvantages, and the most important ones are shown in table 6.1.
<table>
<thead>
<tr>
<th>Method</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Shared Memory</td>
<td>• Packet operations (adding headers and trailers, etc.) can be done easily if packets are stored as linked lists of data fields. Adding a header is then accomplished by allocating a block, filling it with the header data and a few pointer operations to link it into the packet list.</td>
<td>• If multiple grammar processors need access to data in the packet memory, some of them will have to wait until an access port becomes available, which reduces performance. This problem is inherent to shared memory systems.</td>
</tr>
<tr>
<td></td>
<td>• Since all 'transfers' are actually done by passing pointers, they are extremely fast. The pointers can be kept and manipulated in another (physical) memory to reduce the overhead of accessing the actual packet memory to a minimum, so that all bandwidth is available for data I/O.</td>
<td>• Memory management is rather complex (differently sized buffers must be allocated and deallocated for every packet). Combined with the extremely large number of packets that need to be processed (up to about 0.5 million per second) the memory manager design is a serious problem.</td>
</tr>
<tr>
<td></td>
<td>• Although the hardware of a shared memory system is rather complex, it is needed only once or twice (transmitter and receiver separate) for an entire protocol engine, instead of once for every grammar processor.</td>
<td>• The (large) memory banks will have to be physically implemented (with today's technology) using standard dynamic memory cells. The bandwidth of such memories is usually less than that of modern (smaller and static) FIFOs.</td>
</tr>
<tr>
<td>Local Packet Memory</td>
<td>• Requires very simple memory management. Once a packet is processed, its space is automatically released as a side effect of the FIFO operation. The only management required is an overflow detection to prevent the FIFOs from filling up and subsequently losing any additional data.</td>
<td>• Transferring data will decrease performance if the transfer takes more time than the grammar processors need for processing of packets. This effect will become worse when packet size is increased (which is likely to happen). Bus contention is likely to become a problem.</td>
</tr>
<tr>
<td></td>
<td>• Non-blocking access: all FIFOs are accessible simultaneously, because they are physically separate. No access request will have to be deferred.</td>
<td>• The fact that the buffers are physically separate makes it impossible to reallocate free space from one buffer to another where it might be needed. Buffers are fixed in hardware and cannot be shared.</td>
</tr>
</tbody>
</table>

**Table 6.1 Advantages and disadvantages of global and local packet memories**

The best solution would depend on the exact performance demands for a given protocol engine. It is the author's opinion that for more complex engines (more layers, more grammar processors/layer) and for higher performances, the shared memory
implementation will gradually become the better option. This conclusion is based on the following observations:

- In practice it seems that the average packet size doubles with every ten-fold increase in bandwidth. The optimum packet size also increases as the error rate of the communication network decreases. Protocol engines are meant for networks which are much faster than current ones, and future optic networks will provide extremely low error rates. Packet sizes are expected to increase 4 to 8 times for bandwidths in the GBit/s range.

- To obtain communication speeds over 1 GBit/s, packet data paths must be quite wide using current technology memory cells\(^1\). This makes octet or bit oriented operations on sequentially accessed data from packet FIFOs very difficult and requires more complex hardware. RAM allows random access to the data and therefore offers more possibilities to implement such operations than FIFOs would.

The operations on packets are usually independent of the packet contents and thus of its size, unless these packets must be entirely transferred from one subsystem to another. Also, packet assembly operations are easier when all fields are stored as separate data blocks in memory and linked by pointers. With the FIFO system, packets remain totally packed single streams of data, and it becomes more difficult to maintain this structure when the data path becomes wider, especially when the packet structure is complex, as in many of today's protocols. The shared memory system therefore tends to be more advantageous than the local memory system for such protocols. However, for protocols with very small packet sizes (such as ATM) as well as for some protocols that were especially designed to enable highly efficient implementations and for those that require certain operations to be performed on the packet data itself (checksums) the local memory system might be more suitable, because the operations can be performed on the fly while data is transmitted from one FIFO to the next.

6.1.2 Requirements for the packet management system

The previous section mentioned the problems related to storing, sending and receiving entire packets between successive processing phases controlled by different grammar processors. From the arguments follows that the shared memory system is

---

\(^1\) For example: a single-ported memory with an access cycle time of 20 ns (!) and throughput of 2 GBit/s requires a total I/O bandwidth of 4Gbit/s for the memory. Because one access requires 20 ns every access must read or write 80 bits of data.
probably better for the implementation of general communication protocols. This choice dictates the entire structure of the underlying memory architecture, which will basically have to be a large multi-ported memory bank.

The memory must store packets for all connected grammar processors. Since packets are created and deleted dynamically, this requires allocation and deallocation algorithms for blocks of arbitrary size. These algorithms must be fast to get a high performance. Unfortunately this does not define the complete functionality required for the entire packet management system, because there are other complex operations to be performed in the general case. The remainder of this section will be used to discuss these operations and their consequences for the packet management system.

The OSI reference model for protocols defines a number of so called packet length conversion operations which can be performed in any layer (similar kinds of operations might also appear in other types of protocols). These operations deal with packets that are either too long or too short to be transmitted (efficiently or at all) by themselves. Long packets can be split in smaller parts and a number of short packets can be concatenated into a longer one. At the receiver the inverse operations must be executed to retrieve the original packet(s). All of these operations are controlled or executed by layer entities; in this case that would mean grammar processors. However, since packets are stored outside the grammar processors in a special packet management system, that system will also have to be able to perform those operations under strict control of the grammar processors.

The OSI reference model specifies the length conversion operations in terms of so called Service Data Units (SDUs) and Protocol Data Units (PDUs). See figure 1.4 on page 7 for the meaning of these terms in relation to packets received and sent by layer entities. SDUs are transferred between successive layers and PDUs between peer layers. If a layer N entity wishes to send a data unit to a peer entity (PDU), it will do so by sending that data unit to the next lower level for transmission (now that same data unit is called an SDU). The complete set of length conversion operations is given in table 6.2 and the effects are shown graphically in figure 6.1.

As an example from table 6.2, the operation named blocking in layer N, or (N)-blocking for short, is an operation that appends a number \(k\) of (N)-SDUs resulting in a single (N-1)-PDU which is then sent to a peer entity using the lower layer services (flow direction is down towards lower layers). In figure 6.1c, blocking is shown (arrows pointing down) where 2 (N)-SDUs are merged into a single block and converted to an (N)-PDU after putting a header (PCI) in front of each SDU.
a) No conversion operation  

b) Segmenting/Reassembling  

c) Blocking/Deblocking  

d) Concatenation/Separation  

e) Synchronization/Delimiting

**Figure 6.1** Packet operations for each layer from CCITT Rec. X.200 (1993).

<table>
<thead>
<tr>
<th>(N)-Operation</th>
<th>Flow</th>
<th>Operands</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>concatenation</td>
<td>down</td>
<td>k (N)-PDUs</td>
<td>1 (N-1)-SDU</td>
</tr>
<tr>
<td>separation</td>
<td>up</td>
<td>1 (N-1)-SDU</td>
<td>k (N)-PDUs</td>
</tr>
<tr>
<td>blocking</td>
<td>down</td>
<td>k (N)-SDUs</td>
<td>1 (N)-PDU</td>
</tr>
<tr>
<td>deblocking</td>
<td>up</td>
<td>1 (N)-PDU</td>
<td>k (N)-SDUs</td>
</tr>
<tr>
<td>segmenting</td>
<td>down</td>
<td>1 (N)-SDU</td>
<td>k (N)-PDUs</td>
</tr>
<tr>
<td>reassembling</td>
<td>up</td>
<td>k (N)-PDUs</td>
<td>1 (N)-SDU</td>
</tr>
<tr>
<td>delimiting*</td>
<td>down</td>
<td>1 (2)-PDU</td>
<td>k (1)-SDUs</td>
</tr>
<tr>
<td>synchronization*</td>
<td>up</td>
<td>k (1)-SDUs</td>
<td>1 (2)-PDU</td>
</tr>
</tbody>
</table>

(*) Layer 2 only (N=2)

**Table 6.2** Packet length conversion operations indicated in figure 6.1.
Deblocking is the inverse operation (arrows pointing up). There are 8 possible length conversion operations, and the packet management system must be able to perform them all.

SDUs are abstract data entities; they may occur in real implementations but they do not have to. If they do exist, both SDUs and PDUs are basically allocated blocks of data in the packet memory. There is no actual difference between operations on SDUs or PDUs. After consideration of the operations in table 6.2 and figure 6.1 it can therefore be concluded that at the lowest level, all length conversion operations can be performed if the packet management system allows concatenation and splitting of allocated data blocks. Concatenation of multiple blocks can be done by repeatedly concatenating two blocks, and splitting a block in more than two parts can be done by repeatedly chopping off a small part. It is therefore sufficient for the above operations if the packet management system allows concatenation of two blocks into a single one and splitting of a block in two parts at any position.

Packets are sequences of fields, each of which in turn contains an integer number of octets (groups of 8 bits) representing data or control information. Processing a packet usually means that new fields are added in front and/or at the end (transmission) or that the first and/or last fields are removed (reception). Information from these fields is generated/used by the layer entities. Putting a header in front of an existing block is done by allocating space for the header and writing the data provided by a grammar processor in it.

6.1.3 Memory management characteristics

To prevent unnecessary moving and copying of data during these operations, packets are not stored in a single data block in memory, but rather as a linked list of references to data blocks, each representing a field or part of a field. The above operations can all be performed by manipulating these reference lists, which can be kept in a physically separate very fast memory (of a relatively small size compared to the packet storage memory).

There are a few operational characteristics which clearly distinguish this memory management from that of most other (general purpose) systems. An extremely important factor is the asymptotic complexity of the allocation and deallocation algorithms. By extrapolation from current networks (ethernet: 10 MBit/s, 5...10 Kpackets/s, 1000 bits/packet) it follows that for 2 GBit/s networks an average of 250000 to 500000 packets/s of 4 Kbits each is to be expected. If every layer adds one field (requires 1 allocation and 1 deallocation) then a 3 layer engine performs 4 allo-
ocations and deallocations per switched packet, i.e. up to 2 million allocations and 2 million deallocations per second. With the transfer and acknowledge time taken into account it is expected that, on average, many thousands of blocks will be allocated in the system. Even under those circumstances, this speed of allocation must be maintained! Basically this means that the asymptotic complexity of both the allocation and deallocation algorithms must be $O(n^0)$ where $n$ represents the number of blocks already allocated in the memory. No such general algorithm has this property for both operations simultaneously (to the author's knowledge). Fortunately, this case does not need a general algorithm. If the packet size increases by a factor $k$, then the number of switched packets per second will decrease by approximately the same factor, and thus time available for (de-)allocation will increase by factor $k$. In other words, the upper bound on execution time may be proportional to the block size.

Execution time can be reduced by a constant factor using the observation that most of the data blocks will be allocated for fields added by the entities. The format and size of these fields is usually specified by the protocol and thus fixed for each packet type. It is therefore possible to pre-allocate a large number of fields, which will never be deallocated, but reused over and over again for similar packets. Processing a packet for transmission then requires a single allocate and deallocate operation, independent of the number of headers and trailers added.

Another characteristic is that packet buffers are accessed sequentially. This means that it does not matter much if the packet management system maintains rather complex data structures, such that the translation of a block reference plus an offset into an actual memory address is relatively slow. As long as a sequence can be written fast once the first (start) address has been found, performance will still be high.

6.1.4 Primitive functions of the packet management system

A list of primitive operations for the packet management system follows directly from the analysis in the previous sections:

- Allocation and deallocation of blocks of any size
- Concatenation of 2 blocks into a single longer block
- Splitting of 1 block in 2 parts (size specified for the first part)
- Sequentially reading a part of a single block
- Sequentially writing a part of a single block
- Sequentially reading arbitrary sequences of entire blocks (a packet)
- Packet assembly (terminal + attributes $\rightarrow$ data structure)
- Packet disassembly (data structure $\rightarrow$ terminal + attributes)
The packet management system can be divided into two layers: a memory manager and a packet manager. The memory manager performs all low level operations such as block (de-)allocation, block splitting, reading and writing. On top of it, the packet manager performs higher level tasks such as packet creation, disposal, reading, writing, concatenation and splitting. This structure is shown in figure 6.2. Both layers will be discussed and specified in the following sections.

6.1.5 Specification of the memory manager

Allocation and deallocation algorithms require the maintenance of data structures containing information about which memory locations are free and which are allocated. These data structures are usually simple lists of free blocks. Allocation then requires searching this list for a suitable block, which is in the worst case linear in the number of allocated blocks and therefore does not meet the performance requirement. A similar argument holds for lists of allocated blocks, which makes these types of algorithms unsuitable. Searching can be prevented if requested blocks do not have to be contiguous. The requested amount of memory is allocated by gathering a number of smaller free blocks located anywhere in memory. Both allocation and deallocation are then independent of the number of allocated blocks.

Disadvantages are of course that the data structures to be maintained by the memory manager become more complex and that for every access the physical address has to be computed from these data structures first. Because packets are only accessed sequentially this will not be a major problem. The data structures can be designed such that only the first access will be delayed for address computation, but addresses for further sequential accesses can be computed by the memory manager in real time. References to allocated blocks are not physical addresses, but unique identifica-
tion numbers generated by the memory manager which correspond to elements of its data structure. From this data structure the memory manager can compute the actual address of any allocated word in the block.

The memory manager still needs to maintain lists of free blocks. If blocks can start and end anywhere in memory these lists might become very large. To prevent this space is always allocated in units of a so called memory page: the memory manager is designed as a page mode system. This means that the physical memory is partitioned into N disjoint pages of some fixed size K. The minimum allocation unit is a whole page. The total memory size is $K \times N$ and the $n^{th}$ page (zero based count) starts at address $n \times K$. To allocate a block of size S, the memory manager has to find $((S+K-1) \div K)$ free pages, mark them allocated, link them together in a list and assign a unique identification number (so called block reference) to that list.

If an allocated block is split in two parts, then the split will most likely not be on a page boundary. Instead, after executing a split operation, there will be one page which originally belonged to the unsplit block and which is now divided amongst the two parts. If one of the two parts is then deallocated, the split page must remain allocated until the other part is also deallocated. In general, if the page size is $K$ bytes then there can be $K$ different blocks of one byte in a page if no limits are set on the split size. If the split size $V$ is limited to powers of 2 ($V = 2^{m+k}$, where $m$ is some positive integer dictating the minimum split size and $k \in \mathbb{N}$), then the worst case arises when a split occurs every $2^m$ words. This results in $(K + 2^m - 1 \div 2^m)$ allocated blocks whose page list contains the subject page. As a consequence, the memory manager has to maintain a count for every page indicating the number of allocated blocks referring to it. Allocation always consumes empty pages, setting the count to 1. Splitting increments the count, unless it coincides with a page boundary.Deallocation always decrements the count.

A memory manager state MS is a 5-tuple $(K, P, R, C, B)$ where:

- $K \in \mathbb{N}$ is the fixed page size
- $P = \{0, \ldots, c_P\}$ "A finite set of pages (numbers)"
- $R = \{0, \ldots, c_R\}$ "A finite set of block references"
- $C: P \rightarrow \mathbb{N}$ "A function that returns the count for each page"
- $B: R \rightarrow M \cup \{\perp\}$ "A function mapping references to memory blocks (the value $\perp$ is returned for undefined references)"

A memory block $m \in M$ is denoted as a triplet of type $P^* \times \mathbb{N} \times \mathbb{N}$, representing a list of pages occupied by the block, the start offset of the block in its first page and
the total size of the block (physical memory locations). It can have any desired size, be located anywhere in memory and does not have to be contiguous.

For a page list $Q = (q_0, ..., q_i) \in P^*$, element $q_i$ will also be denoted as $Q[i]$. Let $V: N \rightarrow N$ represent the memory function returning the value stored at the location given by the argument. Furthermore, let $L: R \times N \rightarrow N$ be a function that maps a block reference and an offset into a physical memory location (address). Then the $n^{th}$ word of the block whose reference is $r$ is located at address:

$$L(r, n) = K \ast Q[(n+h) \div K] + ((n+h) \mod K);$$

where $B(r) = (Q, h, s)$ and $(n < s)$

The value stored at the $n^{th}$ word of block $r$ is given by $V(L(r, n))$.

If $K$ is taken a power of 2, then this is easy to implement with binary arithmetic. Sequential access to a block is very fast, since addresses can simply be incremented by one for every access. Only when a page border is reached (where $n+h \mod K = 0$), table $Q$ is required to look-up the next page, and this operation be executed in parallel with earlier accesses to avoid unnecessary waiting. Table $Q$ must be implemented as a linked list, because its number of entries (pages) is dynamic.

For convenience in writing, the following notation is introduced:

For a given function $F$, the notation $F[a/b]$ is used to denote a new function which is identical to $F$, except for the argument $a$ where $F[a/b]$ returns the value $b$ independent of what $F$ returned. Similarly $F[a_0/b_0, ..., a_n/b_n]$ is a new function identical to $F$ except for the argument values $a_0, ..., a_n$ where it returns $b_0, ..., b_n$ instead.

The memory manager can execute the following functions:

- **alloc**: allocate a block of a given size
- **dealloc**: deallocate an allocated block
- **split**: split a block in two parts
- **read**: read a number of words of a block from a given offset
- **write**: write a number of words to a block from a given offset

The **alloc** function takes a block size and a memory manager (MM) state and produces a reference to a newly allocated block and an updated MM state.

$$\text{alloc: } N \times MS \rightarrow N \times MS$$

$$\text{alloc}(S, (K, P, R, C, B)) = (r, (K, P, R, C', B'))$$

where: $r \in R$, $B(r) = \bot$, $C' = C[p_{0}/1, ..., p_{n-1}/1]$ with $n = (S + K - 1) \div K$, $B' = B[r/(p_0, ..., p_{n-1}), 0, S]$ with $p_i \in P$ and $\forall 0 \leq i \leq n-1: C(p_i) = 0$
The dealloc function takes a block reference and an MM state and produces a new MM state in which the specified block is no longer allocated.

\[
\text{dealloc: } N \times MS \rightarrow MS \\
\text{dealloc}(r, (K, P, R, C, B)) = (K, P, R, C', B')
\]

where: \( r \in R \), \( B' = B[r/1] \) and \( C' = C[p_0/\text{dec}(C(p_0)), \ldots, p_n/\text{dec}(C(p_n))] \) with \( B(r) = ((p_0, \ldots, p_n), h, S) \) where \( \text{dec: } N \rightarrow N \) is given by:

\[
\text{dec}(i) = \begin{cases} 
  i - 1 & ; i \geq 1 \\
  0 & ; i < 1 
\end{cases}
\]

The split function takes a block reference, a split position and a MM state, then splits the given block at the specified location, finds a new reference for the second part and returns that reference and a new MM state which reflects the new situation.

\[
\text{split: } N \times N \times MS \rightarrow N \times MS \\
\text{split}(r, n, (K, P, R, C, B)) = (t, (K, P, R, C', B'))
\]

where: \( r, t \in R \), \( B(t) = \bot \), \( B(r) = ((p_0, \ldots, p_q), h, S) \), \( 0 < n < S \) and if \( m \) and \( d \) are given by \( m = (n + h) \mod K \) and \( d = (n + h) \div K \) then

\[
(m = 0 \Rightarrow B' = B[r/((p_0, \ldots, p_{d+1}), h, n), t/((p_{d+1}, \ldots, p_q), 0, S-n)] \\
C' = C)
\]

\[
(m \neq 0 \Rightarrow B' = B[r/((p_0, \ldots, p_q), h, n), t/((p_{d+1}, \ldots, p_q), m, S-n)] \\
C' = C[p_0/C(p_d) + 1])
\]

The read and write access functions sequentially read/write data from/into a block. Only the block reference and the initial offset have to be specified. Any number of words can be read or written, but never outside the specified block.

\[
\text{read: } N \times N \times MS \rightarrow N^* \\
\text{read}(r, n, (K, P, R, C, B)) = V(L(r, n)), \ldots, V(L(r, n+z))
\]

where: \( r \in R \), \( B(r) = (Q, h, S) \), \( Q = (p_0, \ldots, p_q) \) and \( 0 \leq n \leq n+z < S \)

\[
\text{write: } N \times N \times N^* \times MS \rightarrow MS \\
\text{write}(r, n, W^*, ms) = ms'
\]

where: \( r \in R \), \( B(r) = (Q, h, S) \), \( Q = (p_0, \ldots, p_q) \),

\[
W^* = (w_0, \ldots, w_q), \ 0 \leq n \leq n+z < S \text{ and } ms' = ms
\]

such that for i: \( 0 \leq i \leq z \): \( \text{read}(r, n+i) = w_i \).
Implementation of these operations is fairly straightforward and will involve tables (arrays) for functions B and C. Substitutions such as $B[a/b]$ then translate into writing value $b$ at location $a$ in the table. Since the page lists are dynamic in size, they will be stored as linked lists, requiring some pointer manipulation to create a new entry or to break one up (split function). Finding unallocated pages (where $C(p) = 0$) for the alloc function can be achieved in a fixed time (!) by maintaining a table of pages with count zero. The front element is then taken whenever a new page is needed. New pages are added when their count reaches zero during deallocation.

From the definitions, it follows that the time for allocation, deallocation and splitting is directly proportional to the number of pages in the block and hence to the total block size (operational requirement). To find unused reference numbers, a strategy similar to that for finding unallocated pages can be employed. A detailed report on the implementation of the memory manager in hardware can be found in [Geurts92].

6.1.6 Specification of the packet manager

The packet manager is the outer shell of the packet management system. It shields the 'users' from the low level memory manager and introduces new operations for packet manipulation. These are of course implemented using the functionality offered by the memory manager.

A packet usually consists of a number of fields containing data or protocol control information. For generated packets these fields will be constructed one by one and inserted in front of or after existing fields. Received packets, on the contrary, have to be stored in one block because the field boundaries are not known until after the packet has been analysed. To discard the first field of a packet, there are two possibilities: split the packet just after the field and deallocate the first part, or maintain a start offset for each packet in a block and adjust the offset. The problem with the first option is that fields are often just a few bytes long, which would create many splits inside a single page. This requires larger counters and is also more complex and takes more time to execute than the second option. Also, actually discarding packet parts might not always be desirable. If the packet has to be routed to another system (e.g. for bridges, gateways and intermediate stations), it might be advantageous to keep it intact. Therefore, a packet in the most general sense consists of a sequence of so called units, where each unit is a part of a block represented by the reference of the block, a start offset in the block and a length.
The packet manager can execute the following functions:

- **pcreate**: create a new packet of a given size
- **pddispose**: delete an entire packet from memory
- **pappend**: insert a packet after another and release its reference number
- **ppprepend**: insert a packet before another and release its reference number
- **psplit**: split a packet in two parts
- **pdiscardheader**: remove a header from a packet without deallocation
- **pdiscardtrailer**: remove a trailer from a packet without deallocation

A packet manager state \( PS \) is a triplet \((MS, W, F)\) where:

- \( MS = (K, P, R, C, B) \) is a memory manager state
- \( W = \{ 0, \ldots, c_W \} \) "A finite set of packet reference numbers"
- \( F : W \rightarrow D \cup \{ \perp \} \) "A function mapping packet reference numbers to actual packets stored in memory"

A packet \( D \) is represented by a sequence \( U^* \) where \( U \in R \times N \times N \) is a unit as mentioned before. Every unit \((r, h, s)\) must be located entirely in a single block:

\[(h \geq 0) \text{ and } (s > 0) \text{ and } (h + s \leq S) \text{ if } B(r) = ((p_o, \ldots, p_o), O, S).\]

The packet manager (PM) operations can now be formalized.

The **pcreate** function builds a new packet consisting of a single field. Its arguments are the field size and a PM state and the result is a packet reference to the created packet and a new PM state to reflect the new situation.

\[
\textbf{pcreate}: N \times PS \rightarrow N \times PS
\]
\[
pcreate(n, (MS, W, F)) = (w, (MS', W, F'))
\]
where: \( w \in W , \ F(w) = \perp , \ \text{alloc}(n, MS) = (r, MS') \) and \( F' = F[w/(r, 0, n)] \)

The **pddispose** function deletes an entire packet from memory, releasing the memory locations occupied by that packet. Its arguments are a reference to the packet and a PM state and it returns a new PM state where all blocks occupied by the given packet are no longer allocated and where the given reference number no longer maps into a packet.

\[
\textbf{pddispose}: N \times PS \rightarrow PS
\]
\[
pdispose(w, (MS, W, F)) = (MS', W, F')
\]
where: \( w \in W , \ F(w) = (u_o, \ldots, u_n) \) with \( n \geq 0 \) and \( \forall 0 \leq i \leq n: u_i = (r_i, h_i, s_i) \in U , \ F' = F[w/\perp] , \ MS' = \text{dealloc}(r_n, MS_n) \) where

\[
\forall 1 \leq i \leq n-1: MS_i = \text{dealloc}(r_i, MS_{i-1}) \text{ and } MS_o = \text{dealloc}(r_0, MS)\]
The **pappend** function takes two packet reference numbers and a PM state and appends the second packet to the first. The reference to the appended (second) packet is released and a corresponding new PM state is returned.

\[
\text{pappend: } \mathbb{N} \times \mathbb{N} \times \text{PS} \rightarrow \text{PS}
\]

\[
pappend(w_1, w_2, (MS, W, F)) = (MS, W, F')
\]

where: \( w_1, w_2 \in W \), \( F(w_1) = (u_0, \ldots, u_n) \) and \( F(w_2) = (v_0, \ldots, v_m) \) with \( u_0, \ldots, u_n, v_0, \ldots, v_m \in U \) and \( F' = F[w_2/\perp, w_1/(u_0, \ldots, u_n, v_0, \ldots, v_m)] \)

The **pprepend** function takes two packet reference numbers and a PM state and inserts the second packet in front of the first. The reference to the inserted (second) packet is released and a corresponding new PM state is returned.

\[
\text{pprepend: } \mathbb{N} \times \mathbb{N} \times \text{PS} \rightarrow \text{PS}
\]

\[
\text{pprepend}(w_1, w_2, (MS, W, F)) = (MS, W, F')
\]

where: \( w_1, w_2 \in W \), \( F(w_1) = (u_0, \ldots, u_n) \) and \( F(w_2) = (v_0, \ldots, v_m) \) with \( u_0, \ldots, u_n, v_0, \ldots, v_m \in U \) and \( F' = F[w_2/\perp, w_1/(v_0, \ldots, v_m, u_0, \ldots, u_n)] \)

The **psplit** function takes a packet reference, a split size and a PM state, then splits the given packet at the given location, creates a new packet reference for the second part and returns both that reference and the PM state for the new situation.

\[
\text{psplit: } \mathbb{N} \times \mathbb{N} \times \text{PS} \rightarrow \mathbb{N} \times \text{PS}
\]

\[
\text{psplit}(w_1, S, (MS, W, F)) = (w_2, (MS', W, F'))
\]

where: \( w_1, w_2 \in W \), \( F(w_2) = \perp \)

\[
F(w_1) = (u_0, \ldots, u_n) \text{ with } \forall 0 \leq i \leq n: u_i = (r_i, h_i, s_i) \in U
\]

Let \( \sigma_k = \sum_{i=0}^{k-1} s_i \) be the total size of the first \( k \) units, and let index \( t \) be such that \( \sigma_t < S < \sigma_{t+1} \) and \( 0 \leq t \leq n \), i.e. the split position is in the unit \( u_t \) at offset \( \delta = S - \sigma_t \). By definition \( 0 \leq \delta < s_t \). Then 2 cases arise:

1. \( \delta = 0 \):
   \[
   F' = F[w_1/(u_0, \ldots, u_{t-1}), w_2/(u_t, \ldots, u_n)] \text{ and } MS' = MS
   \]

2. \( \delta \neq 0 \):
   \[
   F' = F[w_1/(u_0, \ldots, u_{t-1}, v_1), w_2/(v_2, u_{t+1}, \ldots, u_n)] \text{ and }
   \text{split}(r_t, h_t + \delta, MS) = (r', MS')
   \]

   where: \( v_1 = (r_t, h_t, \delta) \in U \) and \( v_2 = (r', 0, s_t - \delta) \in U \)

Note that in case (2) \( u_t \) can be changed into \( v_1 \) and that \( v_2 \) has to be created and linked into the unit list.
The \texttt{pdiscardheader} function takes a packet reference, a header size (byte count) and a PM state, and removes the specified number of bytes from the start of the packet. After the operation, the header contents are no longer accessible and the first byte of the packet is now the one after the header in the original packet.

\texttt{pdiscardheader: } N \times N \times PS \rightarrow PS

\texttt{pdiscardheader}(w, S, (MS, W, F)) = (MS, W, F')

where: \( w \in W \), \( F(w) = (u_o, \ldots, u_n) \) with \( \forall 0 \leq i \leq n: u_i = (r_i, h_i, s_i) \in U \)

Let \( \sigma, t, \) and \( \delta \) be defined as for the function \texttt{psplit}. Then:

\[ F' = F[w/(u_o', \ldots, u_i', u_{i+1}, \ldots, u_n)] \]

where: \( u_i' = (r_i, h_i, 0) \); \( 0 \leq i \leq t-1 \)

\[ u_i' = (r_i, h_i, +\delta, s_i - \delta) \]

Removal of data is implemented by setting the size of the first \( t \) units to 0, and adjusting the \( t+1 \)th unit to remove the remaining \( \delta \) bytes (increasing the start offset and reducing the contents size by \( \delta \)).

The \texttt{pdiscardtrailer} function takes a packet reference, a trailer size (byte count) and a PM state, and removes the specified number of bytes from the end of the packet. After the operation, the trailer contents are no longer accessible and the last byte of the packet is now the one before the trailer in the original packet.

\texttt{pdiscardtrailer: } N \times N \times PS \rightarrow PS

\texttt{pdiscardtrailer}(w, S, (MS, W, F)) = (MS, W, F')

where: \( w \in W \), \( F(w) = (u_o, \ldots, u_n) \) with \( \forall 0 \leq i \leq n: u_i = (r_i, h_i, s_i) \in U \)

Let \( \tau_k = \sum_{i=n-k+1}^{n} s_i \) be the total size of the last \( k \) units, and let index \( t \) be such that \( \tau_t \leq S < \tau_{t+1} \) and \( 0 \leq t \leq n \), i.e. the last \( t \) units and the last \( \delta = S - \tau_t \) bytes of the unit before that must be discarded. Note that \( \tau_{t+1} = \tau_k + s_{n-k} \) and thus it follows that \( 0 \leq \delta = S - \tau_t < \tau_{t+1} - \tau_t = s_{n-t} \). Then:

\[ F' = F[w/(u_o, \ldots, u_{n-t+1}, u_{n-t}', \ldots, u_n)] \]

where: \( u_i' = (r_i, h_i, 0) \); \( n-t+1 \leq i \leq n \)

\[ u_i' = (r_i, h_i, +\delta, s_i - \delta) \]

Removal of data is implemented by setting the size of the last \( t \) units to 0, and adjusting the unit before that to remove the remaining \( \delta \) bytes (reducing the contents size by \( \delta \)).

The discard functions do not deallocate memory. Only the packet disposal function deallocates all units, even if their valid contents size has been set to zero. During any
read or write access, units with zero contents size will be ignored (skipped), which effectively removes these units from the packet. The allocate and deallocate functions pass the request on to the memory manager and perform some adjustments to the data structures of the packet manager. Both operations are still done within a time limit that is proportional to the size of the block and completely independent of the number of allocated and free blocks. The unit lists are dynamic in size, and will therefore be stored as linked lists. The mapping function F can be implemented as a table (array). Concatenation is then possible in a fixed time. Splitting is somewhat more difficult and requires finding the unit where the split should occur. This operation takes an amount of time that is proportional to the split size.

6.1.7 The use of packet references

The two managers in the packet management system are implementable in hardware. The total set of data structures and mappings that have to be maintained for the operations probably make the managers slower and more complex than necessary, but the system of references and mappings also has a number of advantages:

- The use of references to packets removes the responsibility of maintaining locations, offsets, etc. from the users (grammar processors, host system) and that makes operations on packets much easier to implement on these systems.
- Packets can be handed over to other users by just sending them the reference numbers instead of whole lists of numbers (easier and faster).
- Maintaining dynamically growing and shrinking lists (or linked lists) is not possible in the grammar processor, due to its very simple stack oriented internal attribute memory management system.
- An allocated block does not have to be contiguous (the mapping completely abstracts from this), thus allowing (1) better memory utilization, (2) allocation without searching lists for a block of sufficient size, but instead by concatenating free blocks of smaller size (fast and efficient), (3) splitting and concatenation of existing blocks without moving data in memory.
- Complete abstraction from the physical underlying memory organization. As long as the same operations are defined with the same results, the packet management system can be completely changed without having to rewrite any software on the user systems.

Only random access reading and writing are operations which are relatively slow, because for every access the physical location has to be computed first, which requires searching through linked lists. However, as was stated earlier, these opera-
tions should be very rare, as packet fields are read and written sequentially by the packet (dis-)assemblers, the host, and DCE interfaces.

6.1.8 Packet manager interfaces

The packet management system is a global memory and therefore shared between a number of different subsystems of the protocol engine. With respect to the management system, these subsystems will be called the users. Every subsystem that needs to create, delete or change packets is necessarily a user. Not all users may or should perform any operation. For example, a packet assembler may create packets but not delete them. The following list gives all users and the operations they will need to perform in general:

- Grammar processors (all operations)
- Host system interface (allocate, deallocate, read, write)
- DCE interface (allocate, read, write)
- Packet assemblers (PA) (allocate, concatenate, write)
- Packet disassemblers (PD) (discardheader, discardtrailer, read header, read trailer, deallocate)

![Diagram of packet management system](image)

**Figure 6.3** Users of the packet management system and their interconnections.

Figure 6.3 shows how these users connect to the packet management system. This architecture follows logically from the operational requirements. Packet (dis-)assem-
blers need access to packets in the memory. They operate under control of grammar processors, who also interact with the packet management system directly. The host interface stores and retrieves data which has to be transmitted or has been received on behalf of the host. The DCE interface stores incoming packets and sends packets when they are ready for transmission. The interprocessor communications network is just a set of wires used to connect the grammar processors, the host interface and the DCE interface in a way that depends on the protocol to be implemented (chapter 7 will describe how this information is obtained).

If the number of users increases too far the performance will suffer because the operational capacity of the packet management system is limited. This is one of the disadvantages mentioned in table 6.1. For protocols which require many or lengthy operations this is likely to become a bottleneck and perhaps even the limiting factor in achievable performance.

6.1.8.1 Grammar processor command interface

Grammar processors have the final control over all subsystems. The DCE interface and the packet (dis-)assemblers are themselves controlled by grammar processors. The only way for a grammar processor to send commands or receive results is by generating and accepting terminals with attributes. The terminal symbol represents the type of operation (e.g. psplit or pdiscardheader) and the attributes are the arguments (packet references, offsets, etc.). Since the packet manager state (argument and result of all functions) is stored locally in the packet manager, the terminal attributes only represent the remaining function arguments.

One small problem arises for functions that return a result value to the caller, such as psplit. Since commands are issued using output terminals, which can only have inherited attributes, the function result cannot be input to the grammar processor this way. An extra input terminal is required to get function results back. Therefore, functions that do not return any results besides a new packet manager state are represented by a single output terminal and the others by an output terminal followed by an input terminal.

The packet management system must be able to interpret terminal symbols as commands and attributes as arguments, and if necessary, generate a terminal to return a result value in an attribute. Since many grammar processors can be connected to a single packet management system, the values for these terminals must be equal for all of them.
6.1.8.2 Host and DCE interface

The host interface accepts commands from any host system, converts them into commands for the packet management system and sends messages (terminals) to grammar processors to inform them about host actions. Examples are requests to send some data, to read received data, to open or close links, etc. In the first case, the interface would allocate a buffer, store the data from the host in it, and send the reference number as attribute of a terminal corresponding to 'request send data' to a grammar processor of the highest layer. Further processing is then the responsibility of the grammar processors.

The DCE interface is very similar to the host interface. It can store incoming packets in buffers and notify the lowest level grammar processor of such events, and it can read packets that are ready for transmission from the packet memory and send them to the DCE through the front end (see figure 1.7 on page 13). Because packets are read in sequential bursts by the packet manager, they must be moved into a FIFO before they can actually be transmitted (a packet must always be transmitted as a continuous bit stream without any interruptions).

6.1.8.3 Packet assemblers and disassemblers

The packet assemblers and disassemblers help in the construction and analysis of packets. They connect to the packet management system and to a grammar processor. Every packet (dis-)assembler is tailor made for one specific grammar processor and contains information on the format of all packets that can be generated/received by that grammar processor on the channel to which it is connected as well as the output/input terminals with their attributes that are used to represent these packets (this information is entirely specified in the protocol grammars).

The task of a packet assembler is to convert an output terminal from a grammar processor with its attributes into a packet whose type and contents are respectively given by the terminal and the information in the attributes. This requires allocating space to store new packet fields, filling them up with correct data (taken from an internal database and attributes) and/or concatenating them with data fields created in a higher layer. The resulting packet can then be further processed by the same or any other grammar processor. Any possible result code from the packet manager (such as new packet references) must be retrieved from it explicitly using input terminals.
The task of a packet disassembler is precisely the reverse of that of an assembler. Given a packet, the disassembler must read its header and/or its trailer to determine the packet type, convert that into a corresponding input terminal for its grammar processor, extract other relevant information to be stored in attributes and send it to the grammar processor. The header and/or trailer can then be discarded by the disassembler. Should a packet error occur, the whole packet may be discarded and an error output signal must be generated. Since packet types are determined by reading certain bytes in the packet, some assumptions must be made as to the format of these packets (such as how to determine the packet type). Usually, such assumptions can be made if the layer to which that packet belongs is known (if all layer N packets have the same global format, for any N). A packet disassembler built on such assumptions can only analyse packets of a single layer.

6.2 Communication channels

6.2.1 Functionality

Channels connect grammar processors to other subsystems of the protocol engine, including other grammar processors. They can have any width, but since wider channels can transport more bits in parallel they will yield a higher message throughput. A higher performance can be achieved if a sending processor does not have to wait until the receiving processor is ready to accept the message. To make this possible, a channel is buffered at the receiver by means of a FIFO (First In First Out) memory, which also implies that it is unidirectional. If a bidirectional channel is needed, it must be constructed from two unidirectional ones. Channels transport both terminal symbols and their attributes. Since the number and size of all attributes of any given terminal symbol is known and fixed for any implementation, the length of a transported message depends only on the message type, which in turn is determined completely by the terminal symbol (the first word of a message). Another important aspect of channels is that they are always point to point connections, i.e. a channel connects exactly one transmitter to exactly one receiver and this connection is invariable. There is no provision for message broadcasting, or any other kind of multi-endpoint connection.

6.2.2 Channel types

Every device (grammar processor, packet management system, packet assembler, etc.) that must be controlled by or generates data for a grammar processor is accessed by means of a channel. These channels must be defined in the protocol grammar. It
is the mapping of terminals to a unique identification number and a direction (input or output) that defines on which channel every terminal is transmitted resp. received by a grammar processor. The correct creation and connection of channels is a task of the hardware compiler. The language used to define protocol grammars (ProGrIL, see chapter 7) allows the definition of 3 types of channels:

1) Frame channels, which connect a grammar processor output (input) to a packet (dis-) assembly controller. Terminals transported over these channels represent packet types of generated (received) packets. The optional attribute values following the terminal contain control information to be stored in (extracted from) the packet headers by the packet (dis-) assembler.

2) Management channels, which are used to connect two grammar processors. These channels can be used to exchange service primitives between layers, but for all kinds of other purposes as well. A terminal transported over a management channel represents the message type and the attribute values form the message value (contents). A special function of this type of channel is that it has to translate messages while transporting them (explained in section 6.2.3). The interprocessor communication network shown in figure 6.3 consists entirely of management channels.

3) Control channels, which connect a grammar processor to another subsystem (packet management system, FIFO controllers, etc.). An output terminal (with respect to the connected grammar processor) transported on a control channel represents an operation to be performed in the other subsystem, and the attribute values are parameters (arguments) to control the operation. An input terminal usually represents a result of an operation, and the optional attribute values contain additional information.

Each grammar processor is automatically created with precisely as many channels of each type as it will need according to the protocol grammar. The grammar does not specify how these channels should be connected, but only to which channel a certain terminal belongs.

6.2.3 Message translation

Management channels connect grammar processors. Both of them must use compatible alphabets of terminals: if a channel transports data from A to B, then the output terminals of A must match the input terminals of B (as bit patterns). This is accomplished by using translators in all management channels. A translator maps
output terminals of the transmitting grammar processor onto corresponding input terminals of the receiving one. Attribute values will pass unchanged. The translator is basically a look-up table, since the channel endpoints are fixed. The hardware compiler can construct these tables using the information generated by the grammar compiler for each protocol grammar.

6.3 Events and errors

In the top-level overview of the grammar processor (see figure 5.1) there are 2 types of I/O signals, called errors and events, which have remained unexplained. This section will show the use of these signals. Their explanation has been left open, because the functionality corresponding with the use of these signals is related with the operation of entire protocol engines, rather than single grammar processors. It has to do with handling errors and all kinds of exceptional situations. It should be noted here that although provisions have been made to detect some errors and allow handling of exceptions outside the normal execution flow, no research has yet been done to determine exactly which errors and/or exceptions require special care. Some examples will be given in the following sections, but these are by no means the only types of errors or exceptions that can arise.

6.3.1 Exceptional situations

It is possible that one or more grammar processors reach an exceptional state that is caused (introduced) by physical limitations of the hardware implementation, and for which there is no equivalent in the original protocol specification. A good example of such a situation is an overflowing channel buffer, caused by the finite size of these buffers. In the model that has been used so far for these communications it was simply assumed that buffers (input and output tapes of abstract pushdown automaton) were infinite. It is not possible for the grammar processors to interrupt their channel output temporarily and continue with other actions until channel buffer space is again available. Instead, once a communication is started, it must be completed before any other input or output can take place. This is a serious problem because it might cause deadlock. To avoid deadlock in communication there must be some provision by which a grammar processor can know in advance if an output operation can be completed (i.e. if enough channel buffer space is available), and then take measures to resolve the situation.

Another exceptional situation is the handling of 'run-time errors'. These include parse errors, attribute manipulation errors (division by 0, out of range array index-
ing, etc.), situations where all rules for a nonterminal are forever disabled, etc. Most of these are caused by faulty descriptions of protocols (incomplete and inconsistent grammar or erroneous expressions), but instead of just crashing or halting the engine it would be preferred if these situations could be handled gracefully.

### 6.3.2 Exception handling strategies

All kinds of different strategies can be invented to handle these situations. For example, an overflowing channel buffer could simply cause loss of messages. Using an acknowledgment scheme for messages would allow safe communication, but requires hardware and/or software overhead. Another strategy is to add hardware by which the grammar processor can acquire information about the state of the channel buffers, and introduce input terminals that read this information through attributes (via control channels). The grammar should contain explicit tests before doing any output. This places a burden on the grammar constructor and introduces overhead which is unnecessary in normal cases. Errors could simply be ignored (for less serious cases) or reset or halt the machine (out of memory situations). Neither of these strategies is satisfactory. A good strategy would have the following properties:

- Introduce no overhead as long as the situation is normal.
- Allow user definable responses to (handling of) exceptional situations.
- Easy definition and generation of responses in a way that is consistent with the protocol grammar concept.
- Congestion of channels and other exceptions should have a minimal impact on other parts of the protocol engine.

The resulting strategy is called *event handling*, and is in fact very similar to interrupts on normal processors. Events are special nonterminals with no attributes, which have been defined as such in the protocol grammar. Every event is used to represent an exception and every event corresponds to one binary input on the grammar processor (event input). When this input is activated, the grammar processor will first finish processing of the current symbol, then push the corresponding event symbol on top of the parse stack and finally continue normal operation. Since the parse stack now holds the event nonterminal, the further operation of the grammar processor is determined by the production rules that have been defined for the event and the corresponding conditions.

The above strategy applies to the *standard events*. There are actually two more types of events, one of which is the *reset event* which is used to implement entity resetting (a protocol primitive). After unrecoverable errors, it might be necessary to com-
pletely reset part of a running protocol implementation. In a protocol automaton, such a situation requires that the parse stack is cleared (forget whatever was supposed to happen in the future), that all local attribute environments are deleted (since the rule invocation list is now empty) and that the automaton is restarted from a new axiom (reset event nonterminal). Reset events operate in a way similar to standard events, except that before pushing the event nonterminal the stack is cleared, and that the event nonterminal has an action procedure to clear all local environments in the attribute evaluator. The total future behaviour of the automaton is then determined by the rules for the reset event nonterminal. The rules for the event therefore define a complete restart sequence for the grammar processor (note that the global attributes are still available and unchanged after the reset event cycle). The simplest reset rule would be to transform the event into the start symbol of the entire grammar. Standard and reset event nonterminals may also be used in production rules. They will be processed as normal nonterminals, with the exception that just before expanding a reset event nonterminal, the stack will be cleared, as if the corresponding event had actually happened. In other words: there is no difference between externally invoked events and self generated events (using rules).

The third event type is the input event. These events are always associated with one particular input terminal. The grammar processor can be programmed to automatically generate an input event whenever a specific input terminal is present on one of the input channels (as frontmost symbol in the queue). This can be directly done in the grammar for every input event and every rule separately. Since input events are internally generated and indicate the presence of a specific input symbol, they are not allowed to appear in rules. The next paragraphs explain their use.

Sometimes it might happen that an input to a system can occur in almost all states of that system, and that processing of that input must take place as soon as possible (high priority). In a grammar based system, this requires writing rules to accept and process the input from almost every nonterminal. This will not only make the grammar bigger and less readable, but it will also contain many duplicates of the same rules (but for different nonterminals). A change in this behaviour will then require updating all the duplicates. This situation is not very desirable, and input events can help resolve this denotational problem.

Inputs as described above are assigned to an input event in the definition of the grammar. Rules for the event define how the input is processed. By enabling the input event in (a subset of) the production rules of the protocol grammar, the input is automatically handled whenever the grammar processor is working on one of
those rules. No additional rules are required in the grammar. The Input Reader will continuously check all input channels to see if the frontmost symbol on any of these channels has an associated input event, in which case a request is sent to the Pushdown Controller. There the event is handled if it is enabled in the current rule.

6.3.3 Semantics of input and standard events

Input and standard events do not increase the expressive power of protocol grammars. In fact, every protocol grammar with input events can be rewritten into one without these events if the concept of conditions on rules is generalized to include a few additional 'system defined' read-only attributes (such as time). The rewritten grammar will usually be considerably more complex and contain many more production rules than the original.

The set of events which is enabled varies with the rules. Let \( r: X_0 \rightarrow X_1 \ldots X_n \) be a rule, \( SE = \{ SE_1, \ldots, SE_k \} \) a set of standard events, \( IE = \{ IE_1, \ldots, IE_m \} \) a set of input events and \( E = SE \cup IE \). The processing of the RHS of \( r \) with events from \( E \) is denoted \( r \triangleright E \), defined by

\[
   r \triangleright E \equiv X_1 \triangleright E \ldots X_n \triangleright E
\]

where the meaning of \( X_i \triangleright E \) depends on the symbol type of \( X_i \) and is given by one of the transformations defined below.

Consider a protocol grammar \(((V_T, V_N, S, P), A, D, AT, VA, SA), TA, PC)\). Let the standard event inputs be numbered consecutively starting at 1. For each standard event input, a new read-only system attribute is introduced. For the \( j \)th input, it is called \( ev_j \). Each of these predefined attributes is typed boolean and reflects the value of the corresponding binary event input. This will allow rule conditions to depend directly on these binary inputs. Let \( ch: V_T \rightarrow N \) be a function mapping input terminals to input channels, such that \( ch(t) = h \) if and only if \( TA(t) = (input, h) \). Furthermore, let \( in: N \rightarrow V_T \uplus \{ \emptyset \} \) be a function that maps each input channel to the input terminal which is frontmost on that channel (or the empty set if the channel is empty), and let \( ee: V_N \rightarrow B \) be a function indicating if a specific input event is enabled (\( ee(n) \) if and only if \( n \) is an enabled input event). Finally, let \( et: V_N \rightarrow V_T \) be a function that maps each input event nonterminal to its activating input terminal.

**Semantic transformations:**

If \( X_i \) is an input terminal, then \( X_i \triangleright E \) can be replaced by a new nonterminal \( N \) for which the following \( k + m + 1 \) conditional rules must be added to the grammar:
If $X_i$ is an output terminal then $X_i \cdot E$ can be replaced by the sequence $X_i \cdot N$ where $N$ is a new nonterminal for which the following $k + m + 1$ rules must be added:

\[
\begin{align*}
N \rightarrow X_i & \quad \text{cond: } \text{in}(\text{ch}(X_i)) = X_i \\
N \rightarrow SE_1 N & \quad \text{cond: } ev_1 \\
N \rightarrow SE_k N & \quad \text{cond: } ev_k \\
N \rightarrow IE_1 N & \quad \text{cond: } \text{in}(\text{ch}(\text{et}(IE_1))) = \text{et}(IE_1) \land \text{ee}(IE_1) \\
N \rightarrow IE_m N & \quad \text{cond: } \text{in}(\text{ch}(\text{et}(IE_m))) = \text{et}(IE_m) \land \text{ee}(IE_m) \\
N \rightarrow \varepsilon & \quad \text{cond: } \forall p: 1 \leq p \leq k: \neg ev_p \land \\
& \quad \forall q: 1 \leq q \leq m: (\text{in}(\text{ch}(\text{et}(IE_q))) \neq \text{et}(IE_q) \lor \neg \text{ee}(IE_q))
\end{align*}
\]

If $X_i$ is a nonterminal then $X_i \cdot E$ can be replaced by $X_i$, and the following $k + m$ rules must be added to the grammar (if not already present or added):

\[
\begin{align*}
X_i \rightarrow SE_1 X_i & \quad \text{cond: } ev_1 \\
X_i \rightarrow SE_k X_i & \quad \text{cond: } ev_k \\
X_i \rightarrow IE_1 X_i & \quad \text{cond: } \text{in}(\text{ch}(\text{et}(IE_1))) = \text{et}(IE_1) \land \text{ee}(IE_1) \\
X_i \rightarrow IE_m X_i & \quad \text{cond: } \text{in}(\text{ch}(\text{et}(IE_m))) = \text{et}(IE_m) \land \text{ee}(IE_m)
\end{align*}
\]

Application of a rule to expand the nonterminal changes the function $\text{ee}$ in a way that only depends on that rule. When the rule is completed, the old function $\text{ee}$ is restored. This function can easily be maintained by the automaton on a separate stack of bit-vectors, where each bit represents one of the input events ($1 = \text{enabled}$, $0 = \text{disabled}$). A new mask is pushed when a rule is applied and popped when the rule is completely reduced. The top of stack element is the current enable mask.
6.3.4 Using errors and events

The first three properties mentioned in the previous section certainly do apply to the event mechanism. The fourth requires care of the programmer to describe behaviour using rules when a channel becomes blocked. A grammar processor that finds one of its output channels blocked can send control symbols to one or more of its neighbour grammar processors to temporarily stop the data flow until the congestion is resolved, and stop sending symbols to the blocked channel. Errors generated by a grammar processor or any other subsystem are also exceptions. By connecting the error outputs to event inputs, a grammar processor can be used to handle errors in an acceptable manner (for example report them to a host, try to solve a problem and continue if possible).

6.4 An architecture for protocol engines

With the discussion of the packet management system and the communication channels, all elements needed to generate a high level architecture have now been introduced. This section will show one possible architecture of such a protocol engine, but this should by no means be considered the best or only solution. It is just a simple and obvious subdivision of a general protocol in layers, and within each layer into one receiving and one transmitting entity. Basically, grammar processors can be interconnected in an arbitrary way, similar to general multiprocessor systems.

The receiver entities can receive and analyse packets, but not generate any. A single input frame channel connects a grammar processor input to the output of a packet disassembler. A control output and input channel are used to control operation of the packet management system and the disassembler. Furthermore, the grammar processor must be able to communicate with both the upper and higher layer receiver entities and with the transmitter entity of the same layer. This requires three management channels with translators. A typical architecture is shown in figure 6.4. The corresponding transmitter entities are very similar in structure. The only real difference is that these entities can generate packets but not receive and analyse them. A single frame output channel connects a grammar processor to a packet assembler input. The transmitter entity is shown in figure 6.5.

The resulting architecture of an entire protocol engine will be a chain of these receiver and transmitter stages, connected to two physically separate packet management systems, one for the receiver side and one for the transmitter side. This concept is presented in figure 6.6.
Figure 6.4 Structure of a single receiver stage.

Note the symmetry in the design. The chain can continue as far as desired. A combination (RGP; TGP) can also implement only a part of a layer, while the remaining part is implemented in another pair. This can considerably increase performance, if the amount of actual protocol processing is high compared to the extra communication overhead introduced by the division.

Figure 6.5 Structure of a single transmitter stage.
6.5 Conclusion

A possible high level architecture for a complete protocol engine has been given. It consists of a network of grammar processors, a packet management system and packet (dis-)assemblers, interconnected by a network of channels. A choice was made to use a shared memory structure for storing and manipulating packets. The packet management system has been divided in a memory manager layer and a packet manager layer, and the functionality of both layers has been specified formally. The memory management used in this system is such that the time complexity of packet operations is independent of the number of packets stored in the memory, which is vital for obtaining high performance. This system is most efficient when packets are relatively long (1 Kbit or longer) and the protocol does not require operations on the packet data contents (such as CRC computation). In such cases this architecture will be able to implement most protocols and achieve very high performance.
Chapter 7

A Design System for Protocol Engines

The research aim, as stated in chapter 1, consists of the development of a design system for protocol engines. An important part of this system, a general hardware architecture for protocol engines, has been introduced in the preceding chapters. Most of the developments on some other parts of the design system were done at the same time, because of the strong interaction between them and the available hardware modules. This chapter presents the concepts that form the basis for the protocol engine design system, as well as its current status.

7.1 The design path

Generally, a protocol development cycle starts with an idea or some incomplete and often ambiguous description, usually in a natural language (see figure 7.1). This idea is then formalized and verified (if possible) resulting in a protocol specification in a formal language. If the goal is actual implementation, the specification is usually transformed into a suitable description during a number of successive design steps, each of which adds more detail (design choices). To check whether the result of each step still meets the specification, testing or verification is required.

The design system proposed here uses a different approach. The original specification is mapped onto a set of protocol grammars. This mapping can include design choices, so that the protocol grammars actually describe the behaviour of a possible implementation of the protocol, rather than specify the protocol itself. To check whether a set of grammars does indeed describe a system that meets the protocol specification, a verification is required. Once the grammars have been obtained, the remainder of the implementation trajectory is completed automatically.
Any protocol that must be implemented in a protocol engine has to be completely described by a set of protocol grammars. These can be obtained from a specification by extracting layer entity operations and rewriting them in grammar format. Most modern protocols are already (supposed to be) defined in terms of layer entities, which should make this process easier. A semi automatic translation of high level specification languages to grammars might eventually become possible, but this will require extensive research and official international acceptance of a standard specification language in which all protocol standards are defined, and from which automatic implementation can then be achieved.

The steps taken to generate an implementation are shown in figure 7.2. A set of protocol grammars is created from a specification. Since the modules for the final hardware implementation are parametrizable, some of these system parameters can also be extracted from the specification or defined during this phase. Examples of such parameters are the packet memory size and the widths of the buses in the grammar processor (in case these should be fixed). All protocol grammars are separately processed by the grammar compiler, which uses the predetermined parameters to generate the parse tables and microcode needed to run the grammar processors, and a link
information file for the hardware compiler. Every grammar will be implemented in a separate grammar processor.

Not all structural information for the entire engine can be incorporated in the grammars in a logical and elegant way. For example, the interconnection structure (channels) between entities could be extracted from channel and terminal names, but this would enforce a large restriction on the freedom of naming within the different entities (grammars). A separate data structure describing the exact channel connections, as well as the terminal translations (i.e. to which input terminal of the receiver a transmitted output terminal corresponds) is considered a better solution by the author. Another example is the use of events, which requires the connection of event inputs to detectors for the corresponding situation. This information cannot be extracted from the grammars (in the current definition), and adding it to the grammars also does not seem right since most of those situations (and hence the detection hardware) appear in other subsystems, outside grammar processors. Therefore, a separate file containing this kind of information must be created along with the grammars. This structural information file is used only by the hardware compiler.

The final step is the actual generation of the protocol engine architecture. Using the link information file generated by the grammar compiler and the structural information file, the hardware compiler can construct a network of interconnected grammar processors and packet assemblers and disassemblers. The interconnections with
other subsystems, such as the packet memory system also follow from these files. All required modules are taken from a library of architectural parametrizable descriptions, the module library. Parameters are then computed or taken from the information files and substituted in the modules, resulting in a complete protocol engine. Finally the tables (ROMs) are filled with the data and code generated by the grammar compiler.

### 7.2 A specification language for protocol grammars

In the current state, the design system does not contain any tool for the automatic analysis/splitting of protocol specifications or the generation of protocol grammars. These descriptions have to be created manually. As illustrated in section 3.8, the description of protocol grammars in the mathematical tuple format is not very convenient and readable for humans. Therefore, a more comprehensible language for the creation of these tuples has been defined and implemented. It is called ProGrIL (for Protocol Grammar Interface Language), and it allows the definition of a complete protocol grammar by means of a set of plain text language constructs. In addition, it enforces the specification of some more aspects related to protocols, such as the formats of generated and recognizable packets. The latter specifications will be used by the grammar compiler to generate microcode for the packet (dis-)assemblers. See [Bloks93a] for a simple BNF syntax definition with examples, and [Bloks92] for a more comprehensive mathematical semantics definition of ProGrIL in terms of a metagrammar definition. This metagrammar was also used to construct an LL-1 top down parser/compiler for ProGrIL. The parser component checks both syntax and semantics of its input, and the compiler component subsequently generates the necessary parse tables, microcode and other data structures needed for the final hardware implementation.

### 7.3 The module library and hardware compiler

At the time this thesis was written, the design system was not completely implemented. As mentioned in section 7.2, there is no tool for the derivation of sets of protocol grammars from high level specifications. At the lower end, the hardware compiler is still missing. The remainder of this section will present some preliminary ideas for the implementation of the hardware library and its management as well as the hardware compiler.
7.3.1 Library management

The module library is a set of descriptions of the modules that have been mentioned in this thesis (the grammar processor, memory and packet manager, packet assembler, packet disassembler, host & DCE interfaces and channel logic) at register transfer level. It is important to note that these designs must be stored there in a parametrized format, henceforth called an object. When an object is retrieved by the hardware compiler, its parameters must be assigned a value which results in a complete description of an implementable hardware architecture, suitable for processing by a silicon compiler. Parameters can be of different types, depending on their effect on the resulting architecture. Examples are bus width, memory width and depth and the number of inputs or outputs of a certain subcircuit (for scalar type parameters), and logic functions for combinatorial blocks (function type parameters).

The management of the module library is not trivial and will be left to a dedicated software tool (the librarian, not separately shown in figure 7.2) which shields the entire module library from the rest of the system. Management tasks include addition and deletion of objects, retrieval of stored objects when given a set of values to substitute for the parameters, providing lists of objects and their parameters, and optionally translation of internal formats to whatever the hardware compiler can handle (storage format abstraction). Since the designs were created and tested in IDaSS (Interactive Design and Simulation System, see [Verschueren92]), from which it is possible to translate into ELLA or SID (the input format for the ASA silicon compiler by Sagantec), this design file format would be an obvious choice for the hardware compiler output. However, it is not suitable for the library because it cannot hold parametrized circuit descriptions.

7.3.2 Generation of silicon implementations

The hardware compiler is actually a linker. It interacts with the librarian, providing it with the required parameters which it extracts from the link information, structural information and system parameters files and links the resulting designs into one single protocol engine design file. Preferably, this file is in a format from which direct silicon compilation is possible (e.g. SID). When the librarian and the hardware linker are implemented, the entire trajectory from high level descriptions in protocol grammar format to protocol engines in silicon chip layout will be covered in a completely automated process.
7.4 Language and implementation aspects

At this point it is interesting to note the multitude of levels of languages and grammars appearing in the entire design system, as shown in figure 7.3.

A protocol grammar (PG) can be implemented in a protocol pushdown automaton (PPDA). A deterministic PG can be implemented in a deterministic PPDA, an LL-1 PG in an LL-1 PPDA and finally a stable LL-1 PG in a finite LL-1 PPDA. Note that the implementation arrows are one way, and that the 4 categories form a hierarchy. ProGrIL can describe any protocol grammar, but the compiler constructed from ProGrILs' metagrammar will only accept LL-1 protocol grammars. The final intention is that protocol grammars describe the same symbolic language of I/O actions as the original specification, or some sufficient subset of it. The black box implementation model of protocol entities is reflected in the concept of the grammar processor, which is itself an implementation of a finite LL-1 PPDA.
Chapter 8

A Stochastic Performance Analysis Model

One of the main reasons for starting a project on protocol implementations in hardware was the much higher data rates that can be obtained in this way (see section 1.3.2, page 9). Now that a formal basis for a protocol description language and an implementation model for it have been presented, as well as an architecture for complete protocol engines, it is time to examine if higher data rates can indeed be reached. This requires a method for computing the performance of a protocol engine, constructed from a set of protocol grammars.

This chapter will address the problem of finding an estimate of the sustained maximum communication performance obtainable by implementing a protocol using the method outlined in the previous chapters. Before such an attempt is made, it must be established what is meant by 'performance'. The grammar processors only implement the control operations of the protocol, not the operations on packet data. The number of packets per second that can be handled by the grammar processors times the average number of bits per packet must equal the average number of bits exchanged over the medium. Because the second factor can vary widely, the performance is normally expressed as the number of packets handled per second, where all packets have a certain fixed length. It is simply assumed that the packet data path is fast enough to keep up with the control part, or in this particular case, that the grammar processors are not slowed down by the packet management system. Since packets are represented by terminals on frame channels, the performance can be expressed as the average number of terminals (from some specific subset of all terminals) which is accepted or generated per second.

Computation of the actual performance during a specific time interval is extremely difficult and since it is more interesting to have information about the maximum
performance over a long period of time, a simple stochastic model based on protocol grammars has been created which can provide estimates of precisely this information. The maximum performance situation is characterized by the absence of unnecessary waiting, events, time-outs, etc. This also means that inputs are available when needed and that no errors occur. The stochastic model is based on these assumptions to find an expression for the maximum sustained performance. This model and an example of its application to a small part of a protocol grammar for the X.25 protocol shall be presented.

8.1 Concepts of the model

The stochastic performance analysis model is again based on the grammar concept. The idea is basically a hypothetical simulation game of 'playing grammar processor'. Take a grammar processor in some state where it is just beginning to process a new symbol from the parse stack. Simulate the necessary processing steps and note the required execution time. Then repeat this for the next symbol and so on, until a very large number of simulations has been done. Add all execution times, determine how many terminals of some kind were encountered, and divide it by the total time. The result is the average number of terminals of that kind per second during a certain interval after reaching the state in which the first simulation started, i.e. a measure of the performance during that interval.

The problem is that grammar processor states are very complex (they also include attribute values), and that long simulations take much time and computation power. Furthermore, the goal is not the actual performance from some given state, but the average long term performance (independent of specific attributes values). To achieve this goal, the underlying context-free grammar is used as a basis, instead of the protocol grammar itself. Every symbol in every rule is assigned its own fixed execution time, which is computed from the actual implementation mechanism (the finite state machines in the grammar processor) and the grammar compiler output (microcode programs or action procedures for every symbol). Simulation can then be done using the simple context-free grammar and the assigned fixed execution times, which is much easier. However, it does create one problem: the selection of rules for expansion can no longer be computed in each case since all context information (attributes) is gone. Instead it must be estimated what the probability distribution of rules for a given nonterminal is. Simulation then becomes stochastic using this probability distribution.
The resulting stochastic model of the original protocol grammar is so simple that simulation is no longer necessary to find a value for the performance. An analytic method for its computation can be derived from the model directly. The following sections will present the model and the derivation of an expression for the average long term performance.

8.2 Probabilistic context-free grammars

Let $G$ be a context-free grammar $(V_T, V_N, S, P)$ where $P$ contains $k$ ordered and numbered production rules, such that for any nonterminal $N \in V_N$, all rules in $G$ for $N$ have consecutive numbers: $p_i = (N, \alpha) \in P \Rightarrow L_N \leq i \leq H_N$ where $L_N, H_N \in N$ depend only on $N$. Let $\|p\|$ be the number of symbols on the right hand side of rule $p$, i.e. $p = (N, \xi) \Rightarrow \|p\| = |\xi|$. Furthermore, let $\lambda: \{0, ..., |P| - 1\} \rightarrow [0, 1]$ be a probability distribution function over the production rules in $P$ such that:

$$\forall i \in N: 0 \leq i < |P|: (0 \leq \lambda(i) \leq 1) \quad (8.1)$$

and

$$\forall N \in V_N: \sum_{i=L_N}^{H_N} \lambda(i) = 1 \quad (8.2)$$

A pair $(G, \lambda)$ where $G$ is a context-free grammar and $\lambda$ is a probability distribution function as above is called a probabilistic context-free grammar (PCFG). Let $PG$ be a protocol grammar for which $G$ is the underlying context-free grammar, hence $PG = ((G, A, D, AT, VA, SA), TA, PC)$. Furthermore, let $(G, \lambda)$ be a probabilistic CFG where $\lambda$ is chosen (estimated) such that it corresponds to the rule invocation distribution of $PG$. The goal of all this is to let $G$, when used as a language generator with nonterminal expansions chosen randomly according to $\lambda$, imitate $PG$ as closely as possible. Instead of finding an expression for the number of occurrences of some terminal $t$ in the very complex protocol grammar $PG$, it now suffices to find one for $(G, \lambda)$. The only assumption here is that it is possible to estimate $\lambda$ from $PG$.

**Definition 8.1:** NT-T performance of a probabilistic grammar $((V_T, V_N, S, P), \lambda)$.

The expected or mean number of terminals $t \in V_T$ derived from a nonterminal $N \in V_N$ while reducing it to $\varepsilon$ divided by the mean time it takes to reduce $N$ to $\varepsilon$ is called the $(N, t)$ performance of the grammar.
The next sections will establish a model for the computation of the \((N, t)\) performance of a probabilistic grammar for any nonterminal \(N\) and terminal \(t\).

### 8.3 Computation of mean nonterminal reduce time

To compute the average number of occurrences of any terminal \(t \in V_T\), it is necessary to have information about the time it takes to process any symbol in the actual implementation. This time can be determined from the implementation of the grammar processor and the output generated by the ProGrIL compiler. For endmarkers and input and output terminals it equals the time needed to have the symbol processed by the pushdown controller plus the time needed to execute the action procedure minus any overlap between those two parts. For nonterminals the time to find a rule for expansion (execute multiple enable procedures) and perform an actual expansion have to be added. In all cases, the only problem preventing the computation of an exact value for the execution time is that expressions may be conditional and contain loop constructs. In these cases the time can either be given as parametric expression, or an estimate for an average time has to be found. Because multiple occurrences of the same symbol usually have different evaluation expressions, they take different amounts of time to process.

Let the \(j^{th}\) symbol of the \(i^{th}\) rule be denoted \(S_{ij}\) and let the time required to process any symbol \(S_{ij}\) be denoted \(\tau_{ij}\) (the LHS nonterminal has index \(j = 0\)). For nonterminals, \(\tau_{ij}\) does not include searching a rule, expansion and recursively processing the result. These last three operations take a variable amount of time because there is generally more than one rule to choose from, each taken with some probability and requiring a different time to expand and process. Therefore, the time to completely process (reduce) a nonterminal \(N\) to \(\varepsilon\) in a grammar \(G\) will be represented by a random variable \(T_N\), whose mean or expected value \(E_G[T_N]\) is of interest and can be computed. It is given by the following equation:

\[
E_G[T_N] = T_N^{\text{arch}} + \sum_{i = L_N}^{H_N} \lambda(i) \left( T_i^{\text{exp}} + \sum_{j = 1}^{||P_i||} \tau_{i,j} \right) \tag{8.3}
\]

where:
- \(T_N^{\text{arch}}\) is the average time required to find a rule for expansion of \(N\).
- \(T_i^{\text{exp}}\) is the time required to expand a nonterminal using rule \(i\) (constant).
- and \(\tau_{i,j}\) is given by:
The average time to execute a nonterminal $S_{i,j}$ is then

$$\tau_{i,j} = \begin{cases} 
\tau_{i,j} & \text{if } S_{i,j} \in V_T \\
\tau_{i,j} + E_G[T_{S_{i,j}}] & \text{if } S_{i,j} \in V_N
\end{cases} \quad (8.4)$$

The values of $T_N^{\text{rech}}$, $T_i^{\text{exp}}$ and all $\tau_{i,j}$ for $S_{i,j} \in V_T$ can be precomputed or least estimated from the implementation architecture and compiler output. Writing down all equations for the mean reduce time values for all nonterminals therefore results in a set of $|V_N|$ equations with $|V_N|$ free variables. This set of linear equations is (must be) solvable, if the rules of the grammar and their probability distribution function are such that each nonterminal will on average reduce to $\varepsilon$ within a finite number of derivation steps. If the grammar is such that a nonterminal has a finite probability of reducing to $\varepsilon$, but on average more new nonterminals are pushed on the parse stack than completely reduced, the set of equations will yield a negative solution for the mean reduce time of that nonterminal. In this case the nonterminal can be reduced, but that is not expected to happen, considering the probability distribution. If the grammar is such that a nonterminal cannot reduce completely (endless recursion with probability 1) the set of equations will not yield a solution (other than $\infty$) for the mean reduce time of that nonterminal. Because negative expected values imply a potentially infinite growth of the parse stack from the moment the nonterminal first appears on top of the stack, such a result indicates possible instability in the protocol implementation.

Example 8.1 A completely reducible probabilistic CFG

\[
\begin{align*}
V_T & = \{ a, b, c \}, & V_N & = \{ A, B, C \}, & S & = A \\
P = \{ & A \to a \, AB, & \lambda & = 0.2 \\
& A \to a, & \lambda & = 0.8 \\
& B \to b \, C \, c, & \lambda & = 0.1 \\
& B \to b \, c, & \lambda & = 0.5 \\
& B \to A, & \lambda & = 0.4 \\
& C \to a, & \lambda & = 0.7 \\
& C \to B \, a & \lambda & = 0.3
\end{align*}
\]

For simplicity, assume that expansion and rule searching takes zero time, and that a terminal $x$ takes time $\tau_x$, independent of its occurrence with $\tau_a = 10$, $\tau_b = 50$ and $\tau_c = 27$. The equations for the mean reduce times follow directly:
\[ E[T_A] = 0.2 \left( 10 + E[T_A] + E[T_B] \right) + 0.8 \left( 10 \right) \]
\[ E[T_B] = 0.1 \left( 50 + E[T_C] + 27 \right) + 0.5 \left( 50 + 27 \right) + 0.4 \left( E[T_A] \right) \]
\[ E[T_C] = 0.7 \left( 10 \right) + 0.3 \left( E[T_B] + 10 \right) \]

Solving these equations for \( E[T_A] \), \( E[T_B] \) and \( E[T_C] \) yields:
\[ E[T_A] = 27.5, \ E[T_B] = 60 \text{ and } E[T_C] = 28 \]

### 8.4 Computation of mean terminal count

The mean terminal count of some terminal \( t \) from a given nonterminal \( N \) is the expected (average) number of occurrences of \( t \) derived during a complete reduction of \( N \) to \( \varepsilon \). The method used to compute this count is very similar to the one used to compute the mean reduce time. The \( (N, t) \) performance follows by dividing the mean terminal count by the mean reduce time.

Let \( \#t_N \) denote the number of terminals \( t \) encountered in a complete reduction of nonterminal \( N \). For a probabilistic grammar \( (G, \lambda) \), \( \#t_N \) is a random variable whose mean value \( E_G[\#t_N] \) is:

\[
E_G[\#t_N] = \sum_{i=1}^{H_N} \lambda(i) \cdot \sum_{j=1}^{||P||} \kappa_{i,j} \tag{8.5}
\]

where \( \kappa_{i,j} \) is given by:

\[
\kappa_{i,j} = \begin{cases} 
0 & \text{if } S_{i,j} \in V_T \text{ and } S_{i,j} \neq t \\
1 & \text{if } S_{i,j} = t \\
E_G[\#t_{S_{i,j}}] & \text{if } S_{i,j} \in V_N 
\end{cases} \tag{8.6}
\]

Hence \( E_G[\#t_N] \) can be found by solving a set of \( m \) linear equations with \( m \) free variables, with \( 0 < m \leq |V_N| \) depending on the grammar \( G \).

**Example 8.2 Mean number of a’s derived from A in example 8.1.**

For the grammar in the previous example, the mean value of \( \#a_A \) follows from:
\[
\#a_A = 0.2 \left( 1 + \#a_A + \#a_B \right) + 0.8 \\
\#a_B = 0.1 \left( \#a_C \right) + 0.4 \left( \#a_A \right) \\
\#a_C = 0.7 + 0.3 \left( \#a_B + 1 \right)
\]
Solving for \( a_A, a_B \) and \( a_C \) yields:

\[
\begin{align*}
a_A &= \frac{165}{116} = 1.42 \\
a_B &= \frac{20}{29} = 0.69 \\
a_C &= \frac{35}{29} = 1.21
\end{align*}
\]

Hence, the expected number of a's derived in a reduction of A is approximately 1.42. Since the expected reduction time of A was found to be 27.5, it follows that the average number of a's derived per time unit while reducing non-terminal A to \( \varepsilon \) is \( \frac{1.42}{27.5} \approx 0.05 \).

### 8.5 Estimation of the rule search time

In section 8.3, \( T_{N}^{\text{srch}} \) was defined as the average time required to find a rule for the expansion of a nonterminal N. This time includes searching through the rule base, executing any enable condition procedure, testing the system inputs against the rules' look ahead set and processing the result for every rule until one is found that can be applied. To find an expression for \( T_{N}^{\text{srch}} \) without introducing new probability distributions (for the probability that a rules' enable condition evaluates to true) another assumption will be made: in the maximum performance situation a rule will always be found when it is tested for the first time, i.e. none of the rules will have to be tested a second time.

The probability that a specific rule is chosen is already known (defined by \( \lambda \)). If there are \( k+1 \) rules for a nonterminal N and they are numbered \( m \ldots m+k \) then the diagram in figure 8.1 gives the distribution of enable procedure execution times.

Each node labelled \( \tau_i^c \) represents the time to test rule \( i \). The arcs labelled \( \lambda_i \) are selections of rule \( i \) for expansion. By definition, the sum of all \( \lambda \)'s must be 1. Testing the first rule takes \( \tau_m^c \) time units. The probability that it is selected is \( \lambda_m \). If it is not selected, the second rule is also tested. The total time then becomes \( \tau_m^c + \tau_{m+1}^c \) and the probability that this rule is selected is \( \lambda_{m+1} \). It is easy to see that the average value of the rule search time is given by the following formula:

\[
T_{N}^{\text{srch}} = \sum_{i=1}^{H_N} \lambda_i (i) \cdot \sum_{j=1}^{i} \tau_j^c
\]  
\[(8.7)\]
Note that the order in which rules are tested is important. The search stops when a rule is found, which implies that the optimal order of the rules is that of decreasing probability. If the order is not optimal, then rules with low probability but complex enable conditions may have a large effect on the rule search time, which is undesired.

![Diagram](image)

**Figure 8.1 Probability distribution of production rules.**

The model could be extended to allow searches where a rule must be tested more than once by introducing a new variable \( U \), indicating the expected number of unsuccessful searches of all rules. If the sum of all involved enable condition procedure execution times is \( T^u_N \), the rule search time must be increased with \( U \cdot T^u_N \). By substitution of (8.7) in (8.3), the mean nonterminal reduce time equation can be rewritten in the following form:

\[
E_G[T_N] = \sum_{i=1}^{H_N} \lambda(i) \left( T^\text{exp}_i + \sum_{j=1}^{||R_i||} T^e_{i,j} + \sum_{j=1}^{L_N} T^c_j \right) 
\]  

(8.8)

with \( T_{i,j} \) as defined in (8.3). Deriving and solving these equations for a given PCFG and nonterminal-terminal pair can be automated and built into the design system. The only additional information required from the user is the rule probability distribution function \( \lambda \) and perhaps some guiding in the determination of the \( \tau_{i,j} \) for complex attribute evaluation expression (conditional and loop expressions).

### 8.6 Endless protocol systems

Protocol implementations are systems that are never supposed to stop. In the protocol grammar this is reflected by the unlimited right recursion of at least one nonterminal. For convenience in notation, such nonterminals shall be referred to as loop nonterminals. It was already mentioned in section 8.3 that the mean reduce time for these loop nonterminals is infinite. However, it is important to realize that the goal here is to find the average number of occurrences of some terminal per time unit.
while reducing (or trying to reduce) a nonterminal, or in other words the average terminal count of some terminal during one pass through a loop defined by a loop nonterminal, divided by the time to execute that pass.

The basic model can be applied to grammars with loop nonterminals. In order to get a result from the equations, the grammar has to be slightly altered so that \( \infty \) is no longer a solution. This is done in 2 steps:

1) Identify the loop nonterminals. Usually they are known in advance, otherwise they can be found by trying to solve the reduce time equations. Any nonterminal whose reduce time is infinite is either a loop nonterminal or it can derive one.

2) For each nonterminal \( N \) found in step 1, add a rule \( N \to \varepsilon \) to the grammar with probability \( \delta_N \). Of course, the other rules for \( N \) must get a total probability of \( 1 - \delta_N \) (how this is achieved is not critical). These rules are different from the other rules, because the execution time for any operation whatsoever for these rules is defined zero. The only effect of these rules is that they change the values for mean reduce times and mean terminal counts from infinite to some function of all \( \delta_N \).

If the original grammar was \( G \), then the altered one will be called \( G' \). Since each nonterminal in \( G' \) has a finite probability of being reduced, the equations will yield a solution in terms of all added \( \delta_N \). Let \( \delta \) be a vector whose elements are the probabilities of added rules in \( G' \). The relations between the mean reduce time of a nonterminal \( N \) in \( G \) and \( G' \) and mean terminal counts in \( G \) and \( G' \) are then given by:

\[
E_G[T_N] = \lim_{\delta \to 0} E_{G'}[T_N] \quad (8.9)
\]

and

\[
E_G[#t_N] = \lim_{\delta \to 0} E_{G'}[#t_N] \quad (8.10)
\]

For a PCFG (probabilistic context-free grammar) \( (G, \lambda) \) where \( G \) contains loop nonterminals, the NT-T performance can be found by creating a modified PCFG \( (G', \lambda) \) using the above method, solving the requested NT-T performance problem for that grammar and computing the limit when the probabilities of added rules approach zero. Note that this must result in a finite and non-negative value because of the interpretation: a protocol cannot generate an infinite or negative number of terminals per time unit of operation.
Let \( (N, t)_G \) be the average number of terminals \( t \) derived per time unit while reducing \( N \) to \( \varepsilon \) by rules of PCFG \((G, \lambda)\). From (8.9) and (8.10):

\[
(N, t)_G = \frac{E_G[\#t_N]}{E_G[T_N]} = \lim_{\delta \to 0} \left( \frac{E_G[\#t_N]}{E_G[T_N]} \right)
\] (8.11)

This provides a mechanism for the computation/estimation of the sustained maximum performance of a protocol which is described in a protocol grammar. All that remains to be done is the selection of a suitable nonterminal. Since long time average performance is the most interesting, nonterminal \( N \) should be a loop nonterminal. By taking a loop nonterminal, initialization effects are left out of consideration and only the protocol loops starting and ending in the basic state class (see section 4.6.1) are evaluated.

**8.7 The X.25 protocol: an example**

During the development of the grammar processor and ProGrIL, an implementation of the well known and widely used X.25 protocol was made in a protocol grammar to serve as a test case (see [Brouwer92]). It was constructed according to the division principle of layers and receiver/transmitter halves, as shown in figure 6.6 on page 142. The 2 layers of the protocol (layer 2 and 3) were both divided in a receiver and transmitter entity, resulting in a total of 4 entities, each described in a separate grammar. These entities must communicate in order to cooperate, which introduces some overhead. The total size of these grammars in ProGrIL format is approximately 25 pages, half of which is used for the declarations of types, symbols, etc. To reduce the size of the example, only the receiver section of X.25 layer 2 will be considered here.

The terminal symbols have been partitioned into 4 classes (see figure 8.2):

- **data_from_L1**: indications from the physical layer to layer 2 receiver
- **to_L2_tx**: messages to layer 2 transmitter
- **from_L2_tx**: messages received from layer 2 transmitter
- **data_to_L3**: indications sent from layer 2 to layer 3

Attributes, evaluation expressions and conditions can be replaced by a rule probability distribution function, and the processing times for all symbols which are of interest must be computed or estimated.
The diagram in figure 8.3 shows how the processing times of symbols can be computed for the current implementation of the grammar processor. A path through the diagram must be followed from the input corresponding to the symbol type (EM =
endmarker, IN = input, OUT = output, NT = nonterminal), and whenever a fork is encountered, the route whose label is matched must be followed (ap = 'symbol has action procedure', cond = 'rule has condition procedure', enabled = 'result of condition evaluation is true', alloc = 'rule has allocation procedure'). All values in rounded boxes encountered on such a path must be added. The result is the total execution time in clock cycles of the system clock.

![Figure 8.3 Symbol processing timing diagram.](image)

The diagram was created using the maximum performance assumption, so it is not absolutely correct for all possible situations (since no waiting states for inputs are included, no events occur, etc.). Overlap between operations of different parts of the grammar processor have been taken into account, which makes the diagram very accurate. The values of $\tau_{ap}$, $\tau_c$ and $\tau_A$ are the execution times of the action, condition, and allocation procedures by the attribute evaluator for the symbol or rule being processed. These values can be computed or estimated from the output of the ProGrIL compiler, thereby taking into account the actual instruction timing and pipeline effects (conflicts) of the implementation.
The diagram actually gives a somewhat pessimistic view of the symbol processing time because whenever a procedure is started at the attribute evaluator, the rest of the grammar processor can continue until another procedure must be executed, while the diagram was created under the assumption that the grammar processor always waits for a procedure to complete. That assumption leads to precise results only if all symbols have action procedures, which is not generally true.

To avoid unnecessary computations, a selection for a nonterminal and terminal are made for which to compute the NT-T performance. The data transfer phase (after a link has been activated) is described by the nonterminal named InfoTransfer. Since time-outs and congestion are not considered to appear in the maximum performance situation, only rules 4, 6, 9, 10 and 11 remain of interest (the others are used for link up/down, congestion handling and time-outs). Note that rules 5, 7, 8 and 12 have to be included, since they will also be considered by the pushdown controller and their enable procedure will be executed. Therefore, these rules can have an effect on the rule search time even if they are never applied. If the condition expression evaluation time is indicated by a C, allocation time by an A and general action procedure execution time by P and zero means absence of a procedure, the result is:

4: InfoTransfer → RecInfo CheckInfo InfoTransfer  
    C = 0, A = 9, P = 16, P = 11, P = 8
5: InfoTransfer → RecRR L2_Sync_Sender InfoTransfer  
    C = 0
6: InfoTransfer → RecREJ L2_Sync_Sender L2_REJ_Rec InfoTransfer  
    C = 0, A = 9, P = 9, P = 8, P = 10, P = 9
7: InfoTransfer → RecRNR L2_Sync_Sender L2_RNR_Rec InfoTransfer  
    C = 0
8: InfoTransfer → L2_TimeOut TimeOutRec  
    C = 0
9: CheckInfo → L2_SendREJ  
    C = 12, A = 9, P = 16
10: CheckInfo → L2_Sync_Sender  
    C = 10, A = 9, P = 22
11: CheckInfo → L2_Sync_Sender L3_Data_Waiting  
    C = 13, A = 9, P = 8, P = 21
12: CheckInfo → L2_TimeOutFini L2_Sync_Sender L3_Data_Waiting  
    C = 17

Assume the following distribution:
\[
\lambda(4) = 0.99, \lambda(6) = 0.01, \lambda(5) = \lambda(7) = \lambda(8) = 0 \\
\lambda(9) = 0.02, \lambda(10) = 0.03, \lambda(11) = 0.95, \lambda(12) = 0
\]

This means that 2% of incoming packets is rejected, 1% of incoming packets is a reject frame from the other site, and 5% of incoming info packets does not contain data, but only control information. In this case, an interesting terminal is RecInfo,
which represent the incoming info packets, so the problem is to determine the
\((\text{InfoTransfer}, \text{RecInfo})\) performance of this protocol implementation.

For best results, the rules should be reordered for decreasing probability. In that case,
the unused rules \((\lambda = 0)\) can be omitted, since they would be the last rules consid-
ered but never chosen (one of the rules with \(\lambda > 0\) is selected the first time it is
tested). Furthermore, \(\text{InfoTransfer}\) is a loop nonterminal so a new rule must be added
with probability 8:

0: \(\lambda = 0.99-8\):
\[ \text{InfoTransfer} \rightarrow \text{RecInfo CheckInfo InfoTransfer} \]
\[ C = 0, A = 9 \quad P = 16 \quad P = 11 \quad P = 8 \]

1: \(\lambda = 0.01\):
\[ \text{InfoTransfer} \rightarrow \text{RecREJ L2\_Sync\_Sender L2\_REJ\_Rec InfoTransfer} \]
\[ C = 0, A = 9 \quad P = 9 \quad P = 8 \quad P = 10 \quad P = 9 \]

2: \(\lambda = 8\):
\[ \text{InfoTransfer} \rightarrow \epsilon \]

3: \(\lambda = 0.95\):
\[ \text{CheckInfo} \rightarrow \text{L2\_Sync\_Sender L3\_Data\_Waiting} \]
\[ C = 13, A = 9 \quad P = 8 \quad P = 21 \]

4: \(\lambda = 0.03\):
\[ \text{CheckInfo} \rightarrow \text{L2\_Sync\_Sender} \]
\[ C = 10, A = 9 \quad P = 22 \]

5: \(\lambda = 0.02\):
\[ \text{CheckInfo} \rightarrow \text{L2\_SendREJ} \]
\[ C = 12, A = 9 \quad P = 16 \]

The symbol processing times now follow from the diagram in figure 8.3 and the pro-
cedure execution time values given in the grammar.

For terminals and endmarkers:
\[
\tau_{i,j} = \begin{cases} 2 & ; \text{no action procedure} \\ \tau_{ap} + 1 & ; \text{action procedure} \end{cases}
\]

For a nonterminal \(N:\)
\[
\tau_{i,j} = \begin{cases} 0 & ; \text{no action procedure} \\ \tau_{ap} - 1 & ; \text{action procedure} \end{cases}
\]
\[
\tau_i^c = \begin{cases} 3 & ; \text{no condition procedure} \\ \tau_c + 4 & ; \text{condition procedure and } i = L_N \\ \tau_c + 6 & ; \text{condition procedure and } i > L_N \end{cases}
\]
\[
T_i^{exp} = \begin{cases} 3 + \|p_i\| & ; \text{no allocation} \\ \max(1, \tau_A - 1 - \|p_i\|) + 3 + \|p_i\| & ; \text{allocation} \end{cases}
\]
Note that none of the production rules involved here needs an endmarker. For all rules, the operations of the endmarker can be combined with (appended to) the operations of the last symbol on the right side of the rule.

Using the renumbered rules:
\[
T_0^{\exp} = \max(1, 5) + 6 = 11
\]
\[
T_1^{\exp} = \max(1, 4) + 7 = 11
\]
\[
T_2^{\exp} = 0 \text{ (added rule)}
\]
\[
T_3^{\exp} = \max(1, 6) + 5 = 11
\]
\[
T_4^{\exp} = \max(1, 7) + 4 = 11
\]
\[
T_5^{\exp} = \max(1, 7) + 4 = 11
\]

Apparently, the allocation procedures take so little time that they are completed before the pushdown controller is ready to process the next symbol on the stack.

\[
E_G[T_{\text{InfoTransfer}}] =
\lambda(0)(T_0^{\exp} + \tau_{0,1} + \tau_{0,2} + E_G[T_{\text{CheckInfo}}] + \tau_{0,3} + E_G[T_{\text{InfoTransfer}}] + \tau_0^c) +
\lambda(1)(T_1^{\exp} + \tau_{1,1} + \tau_{1,2} + \tau_{1,3} + \tau_{1,4} + E_G[T_{\text{InfoTransfer}}] + \tau_0^c + \tau_1^c) +
\lambda(2)0
\]

\[
E_G[T_{\text{CheckInfo}}] =
\lambda(3)(T_3^{\exp} + \tau_{3,1} + \tau_{3,2} + \tau_3^c) +
\lambda(4)(T_4^{\exp} + \tau_{4,1} + \tau_4^c + \tau_4^c) +
\lambda(5)(T_5^{\exp} + \tau_{5,1} + \tau_5^c + \tau_4^c + \tau_5^c)
\]

Substitution of the corresponding values leads to:

\[
E_G[T_{\text{CheckInfo}}]
= 0.95(11+9+22+17) + 0.03(11+23+17+16) + 0.02(11+17+17+16+18)
= 59.64
\]

\[
E_G[T_{\text{InfoTransfer}}]
= (0.99-8)(11+17+10+59.64+7+E_G[T_{\text{InfoTransfer}}]+3) +
0.01(11+10+9+11+8+E_G[T_{\text{InfoTransfer}}]+3+3)
or:
E_G[T_{\text{InfoTransfer}}] = \frac{107.64 (0.99 - \delta) + 0.55}{\delta}
\]

The equations for the mean terminal count are simpler:

\[
E_G[\#\text{RecInfo_{InfoTransfer}}] =
(0.99-8)(1+E_G[\#\text{RecInfo_{CheckInfo}}]+E_G[\#\text{RecInfo_{InfoTransfer}}]) +
0.01 E_G[\#\text{RecInfo_{InfoTransfer}}]
\]

\[
E_G[\#\text{RecInfo_{CheckInfo}}] = 0
\]
which results in

$$E_G[\#\text{RecInfo}_{\text{InfoTransfer}}] = \frac{0.99 - \delta}{\delta}$$

Thus, the (InfoTransfer, RecInfo) performance $\eta$ becomes:

$$\eta = \lim_{\delta \to 0} \left( \frac{0.99 - \delta}{\delta} \left( \frac{107.64 (0.99 - \delta) + 0.55}{\delta} \right) \right) = 9.24 \times 10^{-3} \text{ packets/cycle}$$

If the system uses a clock frequency of 40 MHz, the number of RecInfo packets handled per second becomes approximately $3.7 \times 10^5$ which, with a projected average packet length of $8 \times 10^3$ bits leads to a bitrate on the medium of 2.96 GBit/s (provided the data path can handle it). Of course this is not an exact figure, but a reasonable estimate. A better implementation of the grammar processor could probably increase the performance considerably. Especially the execution time of procedures in the attribute evaluator seems to be very important in obtaining high performance.

8.8 Conclusion

The performance analysis model presented in this chapter allows fast and easy computation of average sustained performance of a protocol implementation from a protocol grammar. The only assumption is that a fixed probability distribution function exists for all rules for any given nonterminal. Extensions to the model with varying probability distribution functions should also be possible without complicating the equations too much. The values for execution times needed in this method can be computed by the grammar compiler as it generates output code. In this way, the performance analysis can eventually be done during compilation. If necessary, bottlenecks can then be traced and removed by rewriting the protocol grammars.

Application of the model to a real test case of the X.25 protocol resulted in a very high performance figure, which confirms that the original goal has been reached.
Chapter 9

Conclusions and Directions for Future Work

A research project like the one presented in this thesis has many bits and pieces belonging to all kinds of disciplines and covers a wide area of modern technology. Some parts of it are highly mathematical and abstract, others are concrete electronics designs for which skills in designing complex circuitry are required and again others require programming skills in specific areas such as construction techniques for compilers for a given language and target architecture. Much of the work that had to be done cannot be presented in this thesis, even though it is very important. As an example consider the ProGrIL definition, its metagrammar and its compiler and the exact implementation of the grammar processor down to the lowest level. These designs, tools and documents shall be left to those that will complete the design system. But more importantly, they have already been used and even though they are not included in this thesis, it is still possible to draw conclusions from those experiences.

9.1 History and status of the research project

Although the goal of this research was the development of a general hardware architecture and a design system, the key concepts are definitely the formal development of the protocol grammars and the protocol automaton and of course the translation (implementation) algorithm with its proof. For this reason, these topics have been presented in two full chapters. One should realize that only the implementation in the abstract automaton has been proven. In the real implementation, there are many places where errors may have occurred such as the actual implementation of the protocol automaton (grammar processor) and the ProGrIL compiler which transforms descriptions into protocol grammars and subsequently implements them in a grammar processor.
The general idea for the protocol engine architecture followed not long after the protocol automaton was invented. Since any protocol is divided into a number of cooperating entities, so will the protocol engine when each entity is separately described in a grammar and then implemented. The result is a set of grammar processors which are connected by a set of communication channels. To achieve highest possible performance (which is of course the main reason for this entire research project) each grammar processor will only receive and process the data it needs, and not the total packet contents for each received and/or transmitted packet. The packet management system is located outside the grammar processors and is merely controlled by them. This means that in the protocol grammars it is not possible to use operations on the packet data other than those offered by the packet management system. The fact that symbol processing times are independent of any packet contents or size makes it possible to obtain the extremely high throughput as computed in chapter 8. A disadvantage is that some useful operations, such as CRC computation are not possible, unless dedicated (multiple bit parallel) hardware is built into the packet management system.

The design system is still under construction. Many modules which will eventually form the hardware library have already been designed and tested, but some still have to be developed. At the centre of this system is the ProGrIL compiler which is finished and operational. Ideas for the completion of the system, especially the library management, have been given in chapter 7. When these are implemented, the path to direct silicon compilation from protocol grammars shall be complete.

9.2 Conclusions from the research results

In relation to the original research aim, the following conclusions can be drawn:

- By extending standard context-free grammars with attributes, input and output symbol distinction and conditional rules, it is possible to obtain a formal description model for protocol implementations, called a protocol grammar.
- By extending standard pushdown automata with attribute management and both input and output tapes a protocol automaton is obtained which can be used to implement any protocol grammar. This is mathematically proven.
- Using some restrictions on protocol grammars, a finite deterministic physically implementable version of the protocol automaton can be made. This is called the grammar processor.
- The protocol grammar model and the implementation algorithm were used to construct the grammar processor and a grammar compiler.
Conclusions

• A general architecture for protocol engines has been developed, consisting of a network of grammar processors connected by channels, and one or more packet management systems. All communications in the entire system are done in the same way (terminal symbols with parameters). The precise architecture is not fixed but can vary for each protocol, offering much freedom.

• High throughput is achievable. Using a simple stochastic performance evaluation model, a reasonable estimate can be given for any terminal communication in the system, and hence for packet I/O. Communication speeds over 1 GBit/s are definitely possible.

Some conclusions made from related work (not presented in this thesis):

• To get an implementation of a protocol, ProGrIL appears to be a much easier and more intuitive way of 'coding' than standard programming languages, such as Pascal or C (from: [Brouwer92]).

• It is also possible to generate software implementations from protocol grammars. This allows both hardware and software implementations from the same description, rapid prototyping and easier testing and debugging of protocols before any hardware has actually been built.

• Many additions were made to ProGrIL during this research and the evaluation of the X.25 test case. These have made it much easier to describe many constructs often required for protocols. However, since the test case had to be kept simple due to time restrictions, not all flaws in the original version may have been discovered.

General remarks:

• The advantages of software over hardware in the implementation of data communication protocols have been largely overcome (see section 1.3.1 on page 8).
  - Grammar descriptions are less complex and easier to change than programming language implementations.
  - Special cases can be handled decently and implementation can be done automatically.
  - When a general (i.e. reusable and externally programmable) system is created in microchip form, the cost of production can be kept relatively low.
  - Using software generators, most protocol testing and debugging can be done in software. When finished, a hardware version can be compiled.

• Although this thesis was directed towards protocol implementations, the sys-
tem presented here can easily be adapted for other purposes. The description model and the implementation architecture are well suited for almost any kind of system that is usually modeled by its I/O. Examples are protocol converters and perhaps in the future natural language interpreters.

9.3 Overall conclusions

The work presented in this thesis will enable the construction of communication systems that run existing or new protocols with extremely high data rates (in the range of 1...10 Gbit/s) using current technology. To achieve this high bandwidth, these systems will be implemented in hardware, instead of the usual software implementations where processing power is starting to become a bottleneck. With this hardware implementation, the limiting factor when packets are getting larger is probably not processing power but buffer memory I/O bandwidths.

Because protocols are so complex, a key concept in this work is that the implementations are generated automatically from a formal description of the protocol. This means that error-free prototypes can eventually be created very rapidly and at relatively low cost. A programmable general protocol engine may be built which can be used to test (execute) these protocols without actually making a custom chip, giving very fast design and turn around times.

To create descriptions of protocols a formal language has been developed which is more intuitive and therefore easier to use than most other languages that can be used for this purpose. It has a good mathematical foundation, around which various extensions to the whole system can be created, such as the generation of software implementations and performance analysis tools.

9.4 Recommendations for future work

To finish the design system, the following things remain to be done:

- Develop and make the library management system as depicted in chapter 7.
- Complete the set of modules, parameterize and add them to the library.
- Develop and implement the hardware linker (output to silicon compiler).

A difficult but probably worthwhile related research topic would be the (semi-) automatic conversion of high level specification languages such as LOTOS or SDL to (sets of parallel) protocol grammars. In that case, tools available for these languages
can be used to prove overall protocol correctness, and the grammars can be used to get an implementation.

A very interesting and highly recommended future project is the generation of software implementations from protocol grammars. Using a formal approach and well known compiler construction techniques it must be possible to automatically generate a (concurrent) parser for a set of protocol grammars (in ProGrIL format) which together describe a single protocol. This "parser" accepts/generates only inputs and outputs according to the rules of the protocol grammars and so it is an implementation of the protocol.

A few additions can be made to the entire design system:

- Support of grammars which are not in LL1 form. If possible, grammars are rewritten in LL1 format automatically and then compiled.
- Automatic rewriting of grammars in more efficient ones. Substitutions of rules for nonterminals (on the rhs) can sometimes eliminate those nonterminals and create longer rules (less average overhead). Dead nonterminals (which can never be derived from the axiom) can be removed. Combination of attribute expressions can be (sub-)optimized so that the total number of action procedures or the total execution time is minimized.
- Computation of stack and attribute memory requirements from a grammar, so that a grammar processor can be given an optimal amount of memory (although it must be theoretically possible, it might not be practically feasible, in which case an estimate can probably be derived using a heuristic model).
- Tools to verify completeness, external and internal consistency, protocol stability, etc. This is far from trivial and would probably take years of research. At this point it cannot be predicted if such tools are feasible.

Specific additions/changes to the grammar processor and ProGrIL:

- Support for multiple timers (queueing mechanisms with sorted queues). Now every timer has to be tested in software (rule condition) separately. This is not practical and certainly not fast when a large number of timers is in use. Making sorted queues is currently possible in ProGrIL, but not easy. A possible solution is the implementation of time-outs as events from a timer queue management system in the attribute evaluator.
- Support for inverse lookup tables (CAM cells, associative memories). This is necessary to find link status records from link addresses if a protocol supports
many virtual links, each with its own address (most protocols do). Currently, such lookup functions must be done in software (linear search algorithm). This is possible, but slow and therefore only practical if few links are actually in use (few entries).

- Support for multiple entity instances. This means that one grammar processor would be capable of running multiple grammars on a time sharing basis. A global scheduling and grammar-process swapping mechanism is required to do this. The advantage is that it then becomes possible to run multiple entity instances or even multiple protocols simultaneously on one system. Currently this can only be achieved by adding more grammar processors and this is certainly not dynamic.

From chapter 8 and [Bloks93b], some conclusions can be drawn on how to optimize protocol grammars for efficiency (high speed execution):

- Try to reduce the number of procedures by merging attribute evaluations for different symbols into one procedure whenever possible.
- Try to avoid conditions on rules as much as possible.
- Use as few attributes as possible, especially complex ones such as arrays and structures. Try to keep them word sized (unpacked) and if possible global.
- Avoid complex operations on attributes, such as divisions, complex array indices and bit operations on structure fields.
- Longer rules are more efficient, since each rule search and expansion introduces overhead. In particular, $\varepsilon$-productions are inefficient.
References

Aho72  Aho, A.V. and J.D. Ullman  

Aho86  Aho, A.V. et al.  
Compilers: Principles, techniques and tools.  
Amsterdam: Addison Wesley, 1986.

AMD89a  The SUPER.NET™ family for FDDI.  

AMD89b  The World Network™ catalog.  

Anderson85a  Anderson, D.P. and L.H. Landweber  
Protocol specification by real-time attribute grammars.  
In: Protocol specification, testing and verification. Proc. of the IFIP WG6.1 4th int.  

Anderson85b  Anderson, D.P. and L.H. Landweber  
A grammar-based methodology for protocol specification and implementation.  
In: Data communications. Proc. of the 9th symp., Whistler Mountain, British  

Barnett88  Barnett, R. and S. Maynard-Smith  
Packet switched networks, theory and practice.  

Bloks91  Bloks, R.H.J.  
A protocol engine architecture.  
In: Computer systems. Proc. of the 4th workshop, Amsterdam, Okt. 11, 1991. Ed. by  
P. Hartel and H. Muller.  

Bloks92  Bloks, R.H.J.  
A metagrammar for ProGrIL.  
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven  
University of Technology, November 1992.  
Internal Report no. PRO/EB/9202.

Bloks93a  Bloks, R.H.J.  
ProGrIL: A language for the definition of protocol grammars.  

Bloks93b  Bloks, R.H.J.  
Code generation for the attribute evaluator of the protocol engine grammar processor  
unit.  
References

Bochmann78  Bochmann, G.V.
Finite state description of communication protocols.

Bochmann80  Bochmann, G.V.
A general transition model for protocols and communication services.

Bonatti87  Traffic engineering for ISDN design and planning.

Brouwer92  Brouwer, A.H.A.
Design of an X.25 grammar for the protocol engine.

Burkhardt86  Burkhardt, H.J. et al.
Testing of protocol implementations - a systematic approach to derivation of test sequences from global protocol specifications.

Chapman90  Chapman, N. P.
Defining, analysing and implementing communication protocols using attribute grammars.
Formal aspects of computing (UK), vol. 2 (1990), no. 4, p. 359-392.

Chesson87  Chesson, G.
Protocol engine design.

Cockburn87  Cockburn, A.A.R.
Efficient implementation of the OSI Transport-Protocol checksum algorithm using 8/16-bit arithmetic.

Comer91  Comer, D.E.
Internetworking with TCP/IP; principles, protocols and architecture. Vol. 1.

Daanen87  Daanen, J.M.V.
Design of an X.25 co-processor.

Danthine80  Danthine, A. A. S.
Protocol representation with finite-state models.
Deasington85  Deasington, R.J.

Denning78  Denning, P.J. et al.
Machines, languages and computation.

FIPS87  Guideline for implementing advanced data communication control procedures
(ADCCP): category: hardware, subcategory: data transmission.
FIPS publication no. 87, 26 Sept. 1980.

Fisher88  Fisher, C.N. and R.J. Leblanc
Crafting a compiler.

Fried89  Fried, J.
A VLSI chip set for burst and fast ATM switching.
In: Communications (BOSTONICC/89). IEEE international conference,

Gelli87  Gelli, P.
Evaluation and comparison of three specification languages: SDL, Lotos and Estelle.
Ed. by R. Saracco and P. Tilanus.

Geurts92  Geurts, L.
Een autonoom geheugenbeheersysteem in hardware.
Digital Information Systems Group, Faculty of Electrical Engineering, Eindhoven

Green80  Green, P. E.
An introduction to network architectures and protocols.

Haas85  Haas, O.
Spezifikation von Kommunikationsprotokollen auf der Basis attributierter
Grammatiken. (in German).

Haas86  Haas, O.
Formal protocol specification based on attribute grammars.
In: Protocol specification, testing and verification. Proc. of the IFIP WG 6.1 5th int.

Hansson86  Hansson, H.A.
Automatic implementation of formal descriptions of communication protocols.
In: Protocol specification, testing and verification. Proc. of the IFIP WG 6.1 5th int.
Harangozó, J.
An approach to describing a data link level protocol with a formal language.

Harangozó, J.
Protocol definition with formal grammars.
Liège: Université de Liège. 1978. P. F6-1... F6-10.

Henshall, J. and S. Shaw
OSI Explained: End-to-end computer communication standards.

Jacobs, M.J.F.
Ontwerp van een push down control als onderdeel van een grammatica processor.

Jensen, M.N. et al.
VLSI-architectures implementing lower layer protocols in very high data rate LANs.

Kain, R.Y.
Automata theory: Machines and languages.

Knuth, D.E.
Semantics of context-free languages.

Krishnakumar, A.S. et al.
Translation of formal protocol specifications to VLSI designs.

Level, W.J.M.
Formele grammatica's in linguïstiek en taalpsychologie,
Deel 1: De theorie van formele talen en automaten (in Dutch)

Lewis, H.R. and C.H. Papadimitriou
Elements of the theory of computation.

Liebowitz, B.H. and J.H. Carson
Multiple processor systems for real-time applications.

Linn, R.J. and W.H. McCoy
Producing tests for implementations of OSI protocols.
Lunteren91 Lunteren, J. van
Ontwerp van een attribuut evaluator als onderdeel van een grammatica processor.

Martini89 Martini, P. and M. Rupprecht
Designing high speed controllers for high speed local area networks.

Nissink87 Nissink, P.H.L.M.
Design of an ISDN co-processor.

Partridge90 Partridge, C.
How slow is one gigabit per second?.

Popescu-Zeletin88 Popescu-Zeletin, R. et al.
End-system and gateway architecture in broadband-ISDN.

Rupprecht88 Rupprecht, M. et al.

Salomaa73 Salomaa, A.
Formal languages. (from: ACM Monograph Series)

Schindler79a Schindler, S. and M. Steinacker
A formal specification of an X.25 protocol machine.

Schindler79b Schindler, S. and M. Steinacker

Schwabe81 Schwabe, D.
Formal techniques for the specification and verification of protocols.
Los Angeles: Computer Science Department, University of California, 1981.
Report no. CSD - 810401.
<table>
<thead>
<tr>
<th>Reference</th>
<th>Author(s)</th>
<th>Title</th>
<th>Publication Details</th>
</tr>
</thead>
</table>
Samenvatting

Uitwisseling van data tussen computers over netwerken kan alleen plaatsvinden als de betreffende apparatuur zich houdt aan de regels van een vooraf afgesproken protocol. Zulke protocollen zijn in het algemeen zeer complex en derhalve zijn bijna alle implementaties ervan gemaakt in de vorm van software die geëxecuteerd kan worden op de betrokken apparatuur. Een nadeel van software implementaties is dat ze relatief traag zijn in vergelijking met hardware. Vanwege de toenemende eisen voor steeds snellere data-uitwisseling en het feit dat de netwerktechnologie nu veel hogere bandbreedtes kan aanbieden dan software-implementaties kunnen benutten, zullen nieuwe implementatievormen gevonden moeten worden, bij voorkeur in hardware.

Om de kosten, de ontwikkelingstijd en de foutenkans te verkleinen zouden implementaties automatisch gegenereerd moeten worden. Dit vereist een formele taal waarin protocol implementaties beschreven kunnen worden, en waaruit hardware architecturen algoritmisch afgeleid kunnen worden. De definitie van zo'n taal en een methode voor de generatie van een implementatie ervan vormen het belangrijkste deel van dit onderzoeksproject.

Dit proefschrift beschrijft een mogelijkheid om automatisch hardware-implementaties te genereren vanuit formele hoog-niveau beschrijvingen. Het is gebaseerd op het idee dat een protocol beschouwd kan worden als een specificatie van een symbolische taal die geheel bestaat uit invoer- en uitvoeracties. De toegestane zinnen worden dan gedefinieerd door de grammaticaregels voor de taal. Het concept van formele talen, grammatica's en ermee corresponderende automaten is in de afgelopen decennia zeer ver ontwikkeld. In dit proefschrift wordt uitgegaan van de klasse van context-vrije talen en worden hierop uitbreidingen gedefinieerd in de vorm van attributen (om de protocolvariabelen en andere contextinformatie in op te slaan), tweerichtingscommunicatie (invoer- en uitvoersymbolen) en condities op regels, (om contextafhankelijke verwerking mogelijk te maken) voor zeer complexe gedragsbeschrijvingen, resulterend in een protocol-grammatica die gebruikt kan worden om de symbolische protocoltaal te modelleren. Op soortgelijke wijze wordt ook de pushdownautomaat, die gebruikt kan worden om een willekeurige context-vrije taal te herkennen, uitgebreid met attribuutbeheer en conditionele verwerkingsmechanismen om een abstracte implementatie voor protocol-grammatica's te krijgen, de zogenaamde protocol-pushdownautomaat. Na formeel het concept van de geaccepteerde/gegenereerde taal te hebben gedefinieerd voor zowel de protocol-grammatica als de protocol-pushdownautomaat, wordt bewezen dat elke protocol-
grammatica ook inderdaad kan worden geïmplementeerd in een protocol-pushdownautomaat en een algoritme om dit te verwezenlijken wordt gegeven.

Om tot een fysische implementatie van een protocol-pushdownautomaat te komen zijn de vrij abstracte operaties ervan wiskundig getransformeerd in concrete en direct implementeerbare operaties. Dit leidde uiteindelijk tot de grammatica-processor, een deterministische eindige versie van de protocol-pushdownautomaat die ontworpen en getest is in een simulatie-omgeving en die het belangrijkste sleutelstuk vormt in de hardwarearchitectuur voor protocol implementaties.

Protocolmachines bestaan uit netwerken van onderling verbonden grammatica-processoren, die elk een deel van het totale protocol implementeren (bijv. een laag of onderdeel daarvan). Deze onderverdeling kan in principe willekeurig gekozen worden. De feitelijke data die moet worden uitgewisseld wordt opgeslagen en verwerkt in een apart gedeeld pakketgeheugen waarvan de functionaliteit van het beheer ook in dit proefschrift wordt gespecificeerd en volledig in hardware kan worden geïmplementeerd. Om tot hoge doorvoer van informatie te komen (meer dan 250.000 pakketten per seconde) zijn er speciale methoden voor het geheugen-beheer bedacht. Er is reeds een begin gemaakt met de implementatie.

Er is ook een stap gedaan om het hele ontwerpssysteem voor protocolmachines gebaseerd op deze techniek op te zetten. Dit systeem zal uiteindelijk alle gereedschappen bevatten die nodig zijn om een protocol te implementeren. De grammatica-compiler is af en de hardwarelinker zal over enige tijd klaar zijn. Een converter van hoog niveau specificatietalen (LOTOS) naar protocolgrammaticas zou een interessante uitbreiding zijn, evenals performance-analyseprogrammatuur en een software-implementatiegenerator. Wanneer alles af is, zullen deze gereedschappen de gebruiker in staat stellen zeer snel implementaties van protocollen te maken in hardware en misschien ook in software. In dit proefschrift worden ideeën gegeven voor de verdere implementatie van de onafgewerkte delen.

Ten slotte is er nog een eenvoudig stochastisch model voor protocolgrammatica's ontworpen, waarmee het mogelijk is om schattingen te maken van de haalbare snelheid wanneer een protocol wordt geïmplementeerd met behulp van de grammaticaprocessor. Toepassing op een X.25 testontwerp laat zien dat zeer hoge snelheden inderdaad mogelijk zijn.
Curriculum Vitae

Rudolf Henricus Johannes Bloks

18 september 1964 : Geboren te Eindhoven


Hoofdrichting Regel- en Systeemtechniek
Afstudeerrichting Medische Elektrotechniek

techniek”,
Vakgroep Digitale Informatiesystemen,
Faculteit der Elektrotechniek,
Technische Universiteit Eindhoven
onderwerp: “Specificatie van primitieven voor een algemeen
besturingssysteem voor transputernetwerken”

Vakgroep Digitale Informatiesystemen,
Faculteit der Elektrotechniek,
Technische Universiteit Eindhoven
onderwerp: “automatische implementaties van communicatieprotocolen in hardware”
STELLINGEN

behorende bij het proefschrift

A Grammar Based Approach towards the Automatic Implementation of Data Communication Protocols in Hardware

door

R.H.J. Bloks

Eindhoven, 5 september 1993
1. Door communicatieprotocollen als talen te beschouwen kunnen deze beschreven worden met behulp van een op formele grammatica's gebaseerd model, hetgeen tevens mogelijkheden biedt om tot automatische implementaties van die protocollen te komen. [dit proefschrift]

2. Naarmate een basismodel, zoals een grammaticaklasse, krachtiger en dus expressiever wordt zullen systeembeschrijvingen in dat model simpeler worden; de implementatie en correctheidsbewijsvoering zullen echter complexer worden, waardoor uiteindelijk een compromis tussen eenvoud in beschrijving en eenvoud in implementatie gezocht moet worden. In dit proefschrift ligt het compromis in de keuze van context-vrije grammatica's. [dit proefschrift]

3. Wanneer een protocol met eindig veel toestanden via een protocolgrammatica moet worden geïmplementeerd in een deterministische uitgebreide stackautomaat is een LL(1)-analyse de eenvoudigste en meest intuitieve methode, waarbij tevens gegarandeerd kan worden dat er voor dat protocol minstens één protocolgrammatica is waarvoor de automaat zelf ook begrensd is. [dit proefschrift]

4. Door de introductie van media met zeer hoge transmissiebandbreedtes is de verhouding tussen bereikbare en maximale communicatiesnelheid sterk gedaald, vanwege de relatief achterblijvende rekenkracht van computerapparatuur. Om dit te verhelpen is het noodzakelijk om protocollen meer hardwaregericht te implementeren. [dit proefschrift]

5. Het toepassen van oude theorien op moderne problemen kan soms tot interessante nieuwe inzichten of oplossingen leiden, die anders waarschijnlijk niet gevonden zouden worden.
6. Veel bedrijven en universiteiten laten zich bij de aanschaf van een computer alleen leiden door de initiële aanschafprijs, zonder rekening te houden met latere verliezen ten gevolge van verminderde productiviteit van goedkopere modellen t.o.v. de duurdere.

7. De uitdrukking ‘het gaat bergafwaarts’ behoeft beslist geen negatieve indruk te wekken, gezien de grote belangstelling waarin de ski-sport zich mag verheugen.

8. Bij toenemende complexiteit van systemen zal voor de creatie van implementaties bij voorkeur gebruik moeten worden gemaakt van formele beschrijvingsmethoden en daaraan gekoppelde automatische architectuurgeneratie teneinde de foutenkans en de ontwerptijd kort te houden.

9. De volgende generatie vertaalcomputers zal zodanig moeten worden geconstrueerd dat ook zinnen als “Deze zin bevat precies vier i’s.” met behoud van semantische correctheid worden vertaald.

10. Kennis van concepten uit de informatica-wereld en het kunnen toepassen daarvan binnen de elektrotechniek moet bij E-ingenieurs sterk worden gestimuleerd.

11. De CCITT aanbevelingen voor protocollen dienen in het vervolg in een formele wiskundige taal te worden uitgedrukt, waarbij tevens de term ‘aanbeveling’ moet worden vervangen door ‘definitie’.

12. Uit het feit dat men door de inzet van neurale netwerken bij de besturing van kerncentrales een veiliger situatie denkt te kunnen creëren kan alleen maar worden geconcludeerd dat het met die veiligheid slecht gesteld moet zijn.