ProGrIL: A Language for the Definition of Protocol Grammars

by
R.H.J. Bloks

EUT Report 93-E-270
ISBN 90-6144-270-2
March 1993
ProGrIL: A language for the definition of protocol grammars

by

R.H.J. Bloks

EUT Report 93-E-270
ISBN 90-6144-270-2

Eindhoven
March 1993
ProGrIL: a language for the definition of protocol grammars / by R.H.J. Bloks. - Eindhoven : Eindhoven University of Technology, Faculty of Electrical Engineering. - Fig., tab. - (EUT report, ISSN 0167-9708 ; 93-E-270)
Met lit. opg.
ISBN 90-6144-270-2
NUCI 832
Trefw.: ProGrIL (programmeertaal) / protocollen (informatica) ; datacommunicatie / protocollen (informatica) ; grammatica.
ProGrIL: A language for the definition of protocol grammars

Abstract

This report is one of the results of the protocol engine research project, which is targeted towards the automatic generation of hardware implementations for modern complex data communication protocols. The word 'automatic' implies the use of computers to accomplish the task which in turn requires a formal language that can be used to describe these protocols. Such a language and a compiler for it have now been created. The language is based on an extension of context-free grammars and is called ProGrIL (Protocol Grammar Interface Language).

This document informally describes the syntax and semantics of the protocol grammar interface language. In addition to defining a protocol grammar, ProGrIL contains some constructs that are necessary to link multiple grammars together in order to obtain a larger cooperating system (see [Bloks93b]). All sections of a ProGrIL description are explained, the exact syntax is given and for all possible constructs a textual explanation is given with some examples to clarify its use. Operator priorities are given, as well as tables with all keywords with their use and meaning. Finally, the concept of time is introduced.

Since there may be no ambiguities or inconsistencies in the protocol description language, its syntax must follow some strict rules, that have been defined in a meta grammar (see [Bloks92]). In principle, this meta description defines the exact syntax of the grammar, but it is not easily readable. Therefore the syntax will be explained in an easier to read and understand form, the BNF notation. Context conditions are explained separately where necessary.

The protocol grammar is used as a formal language to describe complex protocols in such a way, that they can be directly implemented in hardware, but also that formal verification methods are still applicable and interfacing to other high level languages (such as CCS) remains possible.

Keywords: grammar, data communication, protocol

- Bloks, R.H.J.
  ProGrIL: A language for the definition of protocol grammars.
  Eindhoven: Faculty of Electrical Engineering, Eindhoven University of Technology, 1993
  EUT Report 93-E-270.

- Address of the author:
  R.H.J. Bloks
  van Norenburchstraat 2
  5622 KP Eindhoven, The Netherlands
Table of Contents

Abstract ................................................................. iii

Introduction ............................................................. 1

ProGrIL: A language for the definition of protocol grammars  3

1 Grammar sections ...................................................... 3
  1.1 The definitions section ........................................... 5
  1.2 The attributes section ........................................... 8
  1.3 The channels section ............................................. 9
  1.4 The terminals section ........................................... 10
  1.5 The formats section ............................................. 11
  1.6 The events section .............................................. 13
  1.7 The nonterminals section ....................................... 14
  1.8 The axiom section .............................................. 15
  1.9 The productions section ....................................... 16

2 Operators, keywords and time .................................... 23
  2.1 Operator precedence relations ................................ 23
  2.2 Value wrapping ................................................ 23
  2.3 The concept of time ............................................ 24

3 The compiler job language ....................................... 29

4 Configuration files ................................................. 32

Bibliography ............................................................ 35
Introduction

The exchange of data between computers is governed by common rules dictating the format and meaning of message units (packets) used for the communication. Two computers can only exchange data if they use the same set of rules, which is usually referred to as a communication protocol. Modern protocols that are used on large computer networks are flexible and very complex systems, which is why they are usually implemented in software running on the communicating computers themselves. A big disadvantage of software implementations is that they are inherently slower than special hardware solutions. With increasing demands for high speed communication and extremely high speed network technology (using fiber optics) software can no longer utilize available bandwidth and hardware solutions must be found. Because of their complexity this is not a trivial matter and it would be very desirable to have a system that can construct an implementation automatically.

The protocol engine Ph.D. research project is directed towards the automatic creation of a hardware implementation for any protocol specified by a formal description. The basic idea is that once a protocol has been conceived, it is written down or converted into a formal implementation description and then automatically processed, resulting in an output suitable for a low level silicon compiler.

This means that a formal language is required in which protocols can be described. Furthermore, it must be possible to translate any valid description in this language into hardware, which implies that a target architecture and a mapping model from language to architecture (language semantics) must be defined. Obviously, the target architecture and description language are closely related.

The language that was created is based on standard context-free grammars (ref. [Aho72], [Aho86],[Denning78], [Kain72], [Knuth68] and[Lewis81] for the theory of languages, automata and computation). Grammars are mathematical models for the specification of formal languages and can be used to describe the communication language of a protocol. The advantage is that the implementation architecture and semantic model are known (pushdown automaton) and mathematically provable.

In order to allow an easy and practical description of modern protocols, some extensions must be made to the standard context-free grammars (see [Bloks93b], [Haas85] and [Anderson85a] for the reasons and some possible ways of doing this). The resulting protocol grammar has to be formally defined and a language definition
given. Then the implementation architecture (the pushdown automaton, see [Denning78] and [Lewis81]) and the mapping model must also be extended and formally defined. Finally, the correctness of the mapping must be shown. This is all done in [Bloks93b], but not found in any work done by others in this field.

When a communication protocol is defined in the form of a (set of interconnected) protocol grammar(s), it can be translated directly into a (set of) extended pushdown automata interconnected as indicated by the grammars to obtain a system whose external behaviour is precisely as specified by the grammars. Such a system therefore implements the communication protocol. Some specialized protocol functions may be implemented in dedicated hardware units outside the pushdown automata for reasons of efficiency. The abstract version of the extended pushdown automaton is called a protocol pushdown automaton, and its physical implementation is the grammar processor. A protocol engine consists of a network of interconnected grammar processors and some dedicated hardware.

Each grammar processor is 'programmed' for a certain protocol by storing the grammar parse tables and microcode for context changes/updates in its memory. Context changes occur when the protocol uses variables to store information about the past and uses them to guide further actions. The tables and programs are all created automatically by a compiler that derives this information from the protocol grammars. This compiler is part of a protocol engine design system that is currently under development.

One problem with the mathematical definition of the protocol grammars as given in [Bloks93b] is that they are not easily readable by humans. To make grammar definitions easier to understand and use, an intuitive description language is required. The different elements of a protocol grammar are then expressed using constructs from that language. Such a description language has been developed and is called ProGrIL (Protocol Grammar Interface Language). The compiler mentioned in the previous paragraph uses ProGrIL descriptions as its input. The formal definition of its semantics (i.e. the mapping from ProGrIL constructs to protocol grammars is defined in [Bloks92]). This report introduces the ProGrIL language at a less abstract level. It shows the general format of a grammar description and the syntax constructions of the language and their meaning are explained.
1 Grammar sections

One of the design goals was to create a language which is comprehensive and easy to use. Therefore, the protocol grammar interface language will contain a number of keywords. At this moment, there are 60 of them. Their meaning and use will be explained below, and a full list is given in table 2. All protocol grammars start with the keyword `grammar` and end with the keyword `endgrammar`. In between is the body of the protocol grammar, which consists of nine separate sections, that are all prefixed by their own special keyword. These sections must always be present in any protocol grammar and in a fixed order. Although some of the section bodies may be empty, the section keywords must always be present.

Notes:
- The parser for the protocol grammar is not case sensitive. For better clarity, all keywords will be written in *bold italic* characters in this document.
• In the remainder of this chapter, the protocol grammar interface language will be referred to as the protocol language or simply the language, while an instance of it (i.e. a protocol described in the language) will be referred to as a protocol grammar.

The nine sections and their keywords are:

* definitions, which is the definition section. Here the author can define attribute types and constant definitions, similar to the high level programming language Pascal.

* attributes, the attribute declaration section. This section is used to declare all attributes that will be used in the production rules of the grammar.

* channels, the I/O channel declaration section. Here a number of input and output channels can be declared, each with a channel type. These channels can then be used in the terminals section to assign terminal symbols to them.

* terminals, the terminal declaration section. This is where all terminal symbols for the protocol grammar are declared, including their channel and attribute types.

* formats, the frame format definition section. This section defines a frame format for every previously defined terminal assigned to a channel for frames.

* events, the special event nonterminal declaration section. This section defines the names of the nonterminals which correspond to external and input events of the grammar processor (interrupts).

* nonterminals, the nonterminals declaration section of the protocol grammar. Here all nonterminals that will be used in the production rules must be declared, together with their attribute types.

* axiom, the axiom section. This section only contains a single name, which must have been defined in the nonterminals section. The name given here is taken to be the start symbol for the protocol grammar, when execution commences.

* productions, the production rules section of the grammar. This is where all production rules, conditions, and attribute evaluation expressions are defined. There must be at least one rule for every nonterminal declared in the nonterminals section.

Each of the above sections will now be explained in more detail.
1.1 The definitions section

There are only three standard (implicitly defined) types for attributes. These are octet, word, and bool. An octet type variable is exactly 8 bits wide and a word variable has the system word width (any size ≥ 8). A bool type variable can only hold the values true and false, which are predefined constants. The values for words are always considered to be signed, but octet values are not. Therefore, the value range for any octet variable includes all integers within the interval [0, +255], independent of system word size, while word variables in a 16 bit system have a range of [-32768, 32767]. To overcome the limitations of these 3 types, the programmer can define new types which are based on existing types by using constructors. This is very similar to the pascal type definitions. Furthermore, it is also possible to define constants. In the following notation of syntax, the bar ( | ), the square brackets ([ and ]) , and the asterisk ( * ) are metacharacters whenever they do not appear between quotes, with the following meaning: | = OR, [ ] = optional, * = zero or more times. The syntax definition is:

<def_section_body> → [ <const_type_def> ] *
<const_type_def> → <const_def> | <type_def>

To define constants (boolean and signed integer numerical):

<const_def> → <name> = <intbool_const>
<intbool_const> → <znum> | true | false
<znum> → <number> | - <number> | + <number>

Example 1: maxretry = 15
p = true

associates the name ‘maxretry’ with the value 15 and the name ‘p’ with the value true in the rest of the protocol grammar. The names ‘maxretry’ and ‘p’ may not be redefined or used for any other symbol, channel or definition in the same grammar, except as a field name in a structure.

To define new types, a number of constructors are available:
The definitions section

To define a single range of values, the range constructor can be used.

**Example 2:**  
```
receive_counter = < 0, 7>
```

defines the type `receive_counter` to be the range of integer values in the interval from 0 to 7 inclusive. Any attribute that is declared to be of this type can only assume values in this range (see also section 2.2).

To define a list with an enumerable list of named values, use the enum constructor. A list of names separated by commas defines the values for variables of this type. The first value will be mapped to the numeric value 0, the second to 1, etc.

**Example 3:**  
```
size = (small, medium, large, xlarge)
```

defines a type called 'size', whose range is the set of the values small (0), medium (1), large (2) and xlarge (3).

To define complex structures (which combine an ordered set of other types into a single new type), use the structure constructor. Structures are very much like the packed record type in the programming language pascal. An ordered list of names followed by types defines the names by which these so called fields of the structure can be referred to as well as their value ranges.

**Example 4:**  
```
L2_L_ctrl = structure
  I : bool,
  ns : <0,7> ,
  pf : bool,
  nr : receive_counter ;
```

defines the format of the control field in HDLC Info packets.
To define an ordered indexed set with a fixed number of elements, each of the same type (i.e. a one dimensional array), use the array constructor. In an array with \( k \) elements, the first element always has index number 0 and the last element always has index number \( k-1 \).

**Example 5:**

\[
\text{TA1} = \left[ \text{maxretry + 1} \right] \text{bool}
\]

\[
\text{TA2} = \left[ 7 \right] \text{ord(xlarge)} \quad \text{structure a: bool, b: word ;}
\]

defines two arrays named 'TA1' and 'TA2'. The first has 16 elements of type bool, and the second has 7 elements, where every element is itself an array of 3 elements whose type is a structure containing a boolean named 'a' and a word named 'b'.

The expression that is used to denote the number of elements must be computable during compile time (i.e. it must be a constant expression), and it must evaluate into a positive value.

To define another name for a type that has already been declared, the new name can simply be defined as equal to the old name. This is the alias construction:

**Example 6:**

\[
\text{ctl} = \text{L2_l_ctrl}
\]

specifies that 'ctl' is an alternative name for the type 'L2_l_ctrl'

*Note:* Since types must be declared before they can be referred to, even within the definitions section, it is not possible to create recursively defined data structures (either direct or indirect).

The construct `<name>` is used to represent the name of a symbol, an attribute, a channel, a constant or a user defined type. In either case, valid names are:

1) Sequences of 1 or more **letters** and/or **digits**, that start with a letter:
   
   \[
   \text{letter} = \left\{ a \ldots z, A \ldots Z, \_ \right\}
   \]
   \[
   \text{digit} = \left\{ 0 \ldots 9 \right\}
   \]
   \[
   <\text{name}> \rightarrow \text{letter} \left[ \text{letter} \mid \text{digit} \right] *
   \]

2) Sequences of 2 or more characters, that start and end with a quote (').:
   
   \[
   \text{quote} = \left\{ \text{the quote character} \right\}
   \]
   \[
   \text{char} = \text{"the set of all printable characters" \ quote}
   \]
   \[
   <\text{name}> \rightarrow \text{quote} \left[ \text{char} \right] * \text{quote}
   \]

Note that the quotes are actually part of the name in this case.
The construct `<number>` is used to represent numeric constants (values). Three different formats are allowed to represent numbers:

1) Decimal format. A sequence of decimal digits (0..9).
2) Binary format. A `#` character followed by a sequence of binary digits (0..1).
3) Hexadecimal format. A `$` character followed by a sequence of hexadecimal digits (0..9, A..F). This is the only exception to the case insensitivity of the compiler: digits in the range A..F must be uppercase.

### 1.2 The attributes section

Attributes can be considered as variables. They can be used to store information about the history of the system (context information) and also to pass that information on to other parts of the system. Before attributes can be used they must be declared, so that the compiler knows their type and size (it needs this information for the code generation and to do type checking). These declarations are made in the attributes section. There are two main categories: global and local attributes. The difference between the two categories lies in the accessibility and allocation strategy.

Attributes which are declared as global will be allocated space in the attribute memory when the system starts up, and they will remain there for as long as the system runs. Any reference to such a global attribute from anywhere in the system at any time will always refer to this preassigned location. Since global attributes are allocated only once during system start, they can be referred to at any time and from any place within the grammar. It is therefore not necessary to pass global attributes down or up via symbols in the production rules.

Attributes which are declared as local only exist while the production rule for which they were created is being parsed. Space for all local attributes of a production rule is allocated when parsing of that rule is started. As soon as the production rule is completely parsed, this space will be deallocated. If a local attribute named `n` is used in more than one production rule, or if multiple invocations of the same rule can exist, each rule invocation will own and refer to another instance of the attribute `n`. This is similar to local variables in procedures and functions in the language pascal.

Every attribute also has an expression type associated with it. This type defines the range of values the attribute may carry. Allowed types are the 3 standard types, any new type declared in the definitions section, or any complete type construction which would be allowed in the definitions section.
Example 7:  

```
global count, framenr, x : word,
local pf: bool
```

defines 3 globally accessible attributes of type word, and 1 local attribute named pf which is of type bool.

None of the names appearing in the attribute name lists may have been defined elsewhere in either the attributes or definitions section. It is not allowed to use the same name for two different attributes, or for both an attribute and a definition. A similar restriction holds for all declarations made in the terminals, channels, events and nonterminals sections as well.

1.3 The channels section

This section is used to define the communication channels on which the terminals of the protocol grammar are sent or received. The whole protocol engine consists of a number of grammar processors plus additional surrounding interface hardware and buffers. These processors must be able to communicate with each other in order to cooperate. Furthermore, they must be able to control an external packet assembler and disassembler and there must be a way to control or get status information about local hardware, such as buffer memories.

In all cases, these communications are done in exactly the same way: by sending or receiving terminals (tokens) that represent the message type, optionally accompanied by attributes (data) that contain additional information.

Channels are strictly one-way, i.e. they are either an input or an output channel. To distinguish between the 3 different types of communications mentioned above, 3 channel types are defined. Channels for frames are connected to packet assemblers or disassemblers. Any communication over such a channel represents reception or generation of a packet. Channels for management connect different grammar processors. These channels are intended for inter processor communication. Channels for io_control connect grammar processors to local hardware. Through these channels, a processor can control local buffer memory controllers, or retrieve status information about local hardware.
The terminals section

Example 8: up_data: output for frames
           down, up: input for management

defines an output channel named 'up_data' that is supposed to be connected
to a packet assembler (note: assembler because it's an output), and two input
channels named 'up' and 'down' which are supposed to be connected to other
grammar processors, where they should be defined as outputs for management.

Note: as the syntax definition shows, there must be at least one channel definition in
the channel section.

1.4 The terminals section

The terminals section is used to define the set of terminal symbols. These symbols
are the elements of the language we wish to describe. Therefore, every terminal sym-
bol represents either an input or an output action. Inputs and outputs are always
done on channels. Every terminal therefore belongs to a single channel. Terminals
belonging to output channels are called output terminals, and similarly terminals
belonging to input channels are called input terminals. The definition of a terminal
includes its channel.

Output terminals represent output actions. The exact values that have to be gener-
ated may depend on the context of the output symbol and other system parameters.
Therefore, output terminals are allowed to have inherited attributes. The values of
these attributes are computed before the output terminal is parsed and successively
transmitted on the corresponding channel by the attribute evaluator. Since output
actions do not acquire information it is not necessary to allow output terminals to
have synthesized attributes as well.

Input terminals represent input actions. The symbol itself represents reception of a
certain type of packet, a message or status information. This notation is not suffi-
cient, since packets may contain additional information meant for protocol or hard-
ware control. In case of a packet, this information will be extracted by the packet
disassembler. In other cases it is received directly over the channel. The information
is made available to the processor by means of synthesized attributes for these input terminals. The additional information stored in these attributes is received by the attribute evaluator, and is therefore available to the processor as soon as the input terminal has been accepted. Input terminals are not allowed to have inherited attributes, since these are not required for inputs.

The terminal section consists of one or more lists of terminal names followed by a channel name and an ordered list of attribute modes and types for the attributes associated with each terminal and optionally the keyword `discard`. When the discard option is specified for an input terminal, that terminal is automatically discarded if it is present at the system input at a time when the system cannot process that terminal (because it was expecting another terminal on that particular input channel). It is not possible to specify the discard option for output terminals. Whether inputs are discarded or not, the system will always signal an error in such cases.

```
<term_section_body>  →  <terms> [ , <terms> ] *
<terms>          →  <name_list> : <name> ( <symattrlist> ) <tdisc>
<symattrlist>   →  ε | <symattr> [ , <symattr> ] *
<symattr>      →  <attrmode> <tdef>
<attrmode>      →  inherited | synthesized
<tdisc>        →  ε | discard
```

**Example 9:**  a, b: up_data (inherited word),
                c: down() discard,
                d: down(synthesized word, synthesized bool)

declares 2 output terminals named 'a' and 'b' which both have one inherited attribute of expression type word, to be used on the (output-) channel 'up_data', an input terminal named 'c' that does not have any attributes at all, to be used on the (input-) channel 'down', and an input terminal 'd' with 2 attributes on the same channel. Input 'c' may get discarded automatically.

Note that there must always be at least one terminal in each grammar description. It is not allowed to leave the terminal section empty, because a grammar with no terminal symbols can not define any non-empty language.

### 1.5 The formats section

The formats section is used to define frame formats for all terminals assigned to `frames` channels. As stated before, these terminals represent transmission or recep-
tion of a frame on output respectively input channels. The exact formats of these frames and the relation between frame fields and terminal attributes has to be established before frames can be assembled or disassembled by hardware.

Naming conventions are used according to OSI standards. If a layer N+1 grammar processor wishes to transmit some data, this data is stored in main memory as an (N)-SDU (Service Data Unit), and an SDU reference value is passed to the layer N processor (the data itself never passes through the processors). The layer N processor can then add header and/or trailer information, thereby transforming the (N)-SDU into a (N)-PDU (Protocol Data Unit), which can be sent to the next lower layer where it is treated as a (N-1)-SDU.

All terminals on frames channels must have exactly one associated format definition in this section. A definition consists of the keyword pdu, followed by the terminal name to which the definition applies, an ordered list of attribute names, an '=' character and finally an ordered list of frame fields. The list of attribute names may contain only names defined in the attributes section and whose type matches the attribute type definition in the terminal declaration. In the following format definition, these names are used to refer to the attributes assigned to the terminal. These attributes are output sequentially on the external output of the grammar processor during terminal transmission or input sequentially during terminal reception. The generated code will be such that attributes are transmitted/received on frame channels in the order in which they are consumed in the format definition. If an attribute
is used more than once, it is also transmitted/received more than once. Thus attributes can be used in any order and as many times as necessary in packet format definitions. Recognition of packets must be possible deterministically in one pass (by examining each packet word sequentially from the packet buffer), and since the only fixed bit sequences in packets are the constants, they must define each packet format for all possible packets of a given frame input channel uniquely.

Example 10: \texttt{pdu}_a(-\texttt{count}) = \#10010111 \quad \texttt{sdru}(\texttt{count}) \ 43
\texttt{pdu}_d(+x, +pf) = \#1001 \ (4) \ pf \ 2 \ (3) \ x \ #001010000100111

Defines the packet format for terminal ‘a’ as an octet with value $151_{16}$ followed by data stored under reference number taken from attribute ‘count’ followed by an octet with value $67_{10}$, and it defines the format for terminal ‘b’ as 5 octets: value $1001p010$, where $p$ is the value of boolean attribute pf, followed by high byte of value of ‘x’ followed by low byte of value of ‘x’ followed by an octet of value $40_{10}$ and finally $71_{10}$. Since $d$ is an input terminal, the values for $x$ and pf are extracted from incoming packets when they are recognized from the format definition. To be able to do this in a single pass, all format definitions for terminals on a single input channel must be left-to-right deterministically distinguishable by means of their constant fields.

Transmission order is always from most significant to least significant bit, both on octet level and on attribute level.

1.6 The events section

The events section defines a special set of nonterminals, called events. Whenever a standard event input becomes activated, the corresponding event nonterminal will be pushed on the top of the parse stack as soon as the current symbol has been completely processed. At that moment, the rules for the event as defined in the productions section will determine the behaviour of the processor. Eventually the event is completely processed, at which moment the parse stack will appear exactly as it was before the event nonterminal was pushed and the protocol parsing continues normally. This is similar to the concept of interrupts on standard microprocessors. A reset event is similar to a standard event, except that just before pushing the event nonterminal on the parse stack, the stack is completely cleared. The reset event has an action procedure that will discard all local attributes. An input event is a special type of event that can be generated internally by the grammar processor as soon as a specified input terminal arrives at the input. The input terminal name for this type of event must have been previously specified in the terminals section.
The nonterminals section

The nonterminals section is used to define the nonterminal symbols or help symbols for the grammar. These symbols are intermediate symbols used in the construction of a derivation tree during parsing of a string. During generation of sentences they can also be considered as procedure names that define a certain sequence of actions (input/output) to be executed.

Example 11: up_full, up_ready : standard

defines two special events named 'up_full' and 'up_ready', which could be used to represent the following external hardware conditions: up_full = 'upward data buffer has become too far filled to accept new data' and up_ready = 'upward data buffer has been emptied far enough to allow entering of new data'. The event inputs of the grammar processor must be wired so that these input are activated when these situations are detected, and the production rules in the grammar must be such that after parsing an 'up_full' symbol all outputs to the upward data buffer are deferred until an 'up_ready' is encountered on the parse stack.

Example 12: sys_error: reset

defines a reset event named 'sys_error' that will remove all symbols from the parse stack, and then push a nonterminal with action procedure on the stack which will determine the complete future behaviour of the protocol automaton. The action procedure is executed as soon as processing of the event nonterminal commences and will remove all local attributes in attribute RAM. The global attributes are not affected.

Example 13: priority_packet: input(a)
defines an input event named 'priority_packet' that can be generated internally by the grammar processor (if enabled) as soon as it detects the input terminal 'a' as next input on one of its input channels. Event rules should then handle processing of that input. This mechanism allows simple priority schemes and input handling without explicitly checking for it in every state.

1.7 The nonterminals section
Nonterminals can by definition never appear in any sentence of a grammar. Since all input and output actions are represented by terminals, nonterminals can never be observed at the outside of the system (they do not represent externally observable actions).

Nonterminals are general constructors for the grammar sentences, and as such they are allowed to have both inherited and synthesized attributes. The values of inherited attributes are computed before an attempt is made to find a rule for the expansion of a nonterminal. The synthesized attributes will get a value while that rule is parsed. Inherited attributes can only pass values down, and synthesized attributes can only pass values up. There is no 'bidirectional' attribute mode.

Note that events are also considered nonterminals. The main differences between the two in the grammar language are that events cannot have any associated attributes. Using a reset event in the RHS of a rule will not clear the parse stack!

To expand a nonterminal, the system will look for enabled rules defined for that nonterminal, by using look ahead set matching with available inputs. If system inputs are such that all rules are disabled and can never become enabled again, a deadlock occurs. In this situation, automatic input discarding can be performed on the specified 'discard channel'. Use this with care, since discarding is done without regard for the input terminal itself, and other alternatives can offer better solutions (such as event generation upon deadlock error signals).

\[
\begin{align*}
\text{<nonterm\_section\_body>} & \rightarrow \text{<nonterms>} [ \text{, <nonterms>} ] * \\
\text{<nonterms>} & \rightarrow \text{<name\_list>} ( \text{<symattr\_list>} ) \text{<ntdisc>}
\end{align*}
\]

**Example 14:** $A, B : \text{(inherited word, synthesized bool),}$

$C : () \text{discard(up\_data)}$

defines two nonterminals named 'A' and 'B' which both have 2 attributes, and a third one named 'C' without attributes which will discard inputs on channel 'up\_data' to resolve deadlock situations.

### 1.8 The axiom section

This is an extremely simple section, since it consists of only one name: the name of the nonterminal (defined in the nonterminals section) that will be used as the start symbol (axiom) of the entire grammar. The axiom must be a nonterminal without attributes.
1.9 The productions section

The productions section is where the actual production rules for the grammar are defined. These rules define how any nonterminal symbol may be transformed into a (possibly empty) sequence of other symbols (both terminals and nonterminals). Rules can be regarded as behaviour descriptions (communication action sequences, since each terminal is an I/O event) or as procedures named after the left hand side non terminal and defining a set of I/O actions (right hand side = procedure body).

There must always be at least one rule (for the start symbol). In fact, there must be at least one rule for every declared nonterminal (including the event nonterminals). Not all terminals and nonterminals have to appear somewhere on the right half side (RHS) of any rule, but it is an error if a declared nonterminal has no associated production rule. All symbols and attributes used in the rules must have been declared in the appropriate section, otherwise an error is generated. Finally, all standard nonterminals (defined in the nonterminals section) must be derivable from the axiom.

It is allowed to use so called epsilon productions, whose left hand side (LHS) consists of the empty string, usually denoted by ε. Since this character is not always available on all systems (it is not a standard ASCII character), another character will be used to denote ε: @. For the same reason, the right arrow, which is usually placed between the LHS and RHS section of a rule has been changed into the '=' character. Attributes are placed between parentheses, and preceded by either a '+' character to denote synthesized, or a '-' character to denote inherited attributes. Although this information is already known to the compiler from the declaration of the symbols it is still required to explicitly denote the mode of each attribute to reduce the possibility of errors. The compiler will check this for consistency with the declaration.

The productions section contains a list of one or more production rules. A rule generally consists of three components, but not all three have to be present for every rule. These are the rule, the condition, and the evaluation component. All three components (if present) start with a keyword which equals the name of the component.

The first component (rule) is the production rule itself. This component starts with a nonterminal or event (lhs) followed by a list of attributes, the '+' symbol for readability and either a '@' character denoting an empty production or a list of one or more symbols (rhs), each followed by a list of attributes and finally a sequence of input event names, each preceded by the par symbol ('\|'). If the lhs is an input event, then the rhs must start with the input terminal for which the event is generated.
(unless it is an empty production), and no pars may be specified. No \textit{rhs} may ever contain an input event. The input events specified in the par list are enabled during the rule, all others are disabled. All symbol names must have been declared in the terminals, nonterminals or events section, the number of attributes and the mode (inherited or synthesized) and expression type of all attributes must match that in the declaration. Since this component defines the actual rule, it must of course be present in every production.

The second component \textit{(condition)} defines the enable condition for this particular production rule. This is an expression whose result type is boolean. If the expression evaluates to true, the rule is enabled, otherwise it is disabled. The condition expression may of course depend on all attributes that are known at the time it has to be evaluated. These are the globally defined attributes (since these are always accessible) and the LHS inherited attributes (which are computed before the parser starts looking for a possible expansion rule). The condition component does not have to be present. The absence of a condition automatically defaults to a true condition, so conditions only have to be specified if they are not constant true.

The third component \textit{(evaluation)} defines the evaluation expressions for the alterable attributes that are used in the rule component. The term ‘alterable attribute’ has been defined in the formal introduction of protocol grammars. For every alterable local attribute, there should be one expression defining its value in this section. If such an expression is missing, an appropriate warning message will be generated by the compiler. Similarly, if an attribute (or part of an attribute) has more than one associated assignment expression, a warning will be given. On the other hand, if an assignment expression is found for a local attribute which is not alterable in the corresponding rule, a compilation error is generated.

An assignment expression starts with the name of the destination attribute, then the assignment symbol (':=') followed by the evaluation expression itself. For structure type attributes, every field may have its own assignment, but it is also possible to copy the value of one structure attribute to another (of the same type) using a single assignment expression. The same holds for array elements and whole arrays.

Assignment expressions can be grouped into expression lists. In a list, expressions are separated by semicolons (';'). The last expression in the list is not followed by a semicolon. An expression list can also be conditional. In that case the list is preceded by an 'if<condition> then' construction and followed by the terminator fi. There is also a special loop constructor to write conditional loops. Such a loop consists of an ini-
The productions section

... 

...
tually) in the same evaluation section, and not just one expression.

5) The initial (implicit) default symbol number is 0, which means that (3) has no effect on (1) and (2). Therefore, the default can be restored later in a list by specifying 0 as the symbol number.

6) Expressions from the evaluation section are automatically sorted by symbol by the grammar compiler. There is no guarantee that the order of expressions is maintained. Expressions in an evaluation section can be written in any order. This can give rise to a particular problem, shown here:

```
rule X -> A(-i) B(-i, -k)
evaluation
  i := 1
(2): i := i + 1; k := i * i
```

At symbol 1 (A), i becomes 1. At symbol 2 (B), i becomes 2 but it is not defined what happens to k. It can become either 1 or 4. Clearly the order of execution is important here. Such cases should be avoided.

7) All expression within a conditional list or within a loop are treated as a single 'super expression'. They are all grouped and executed together for one symbol. The order of execution of the expressions in a loop body or loop initialization is defined to be the same as the textual order in the evaluation section. Within a conditional expression list, the order is not defined.

All evaluation expressions must obey the one pass condition. This condition states that an attribute may only be used for the evaluation of another attribute (the destination) if it has already been assigned a value (i.e. if it appears as a non alterable attribute in a symbol to the left of the symbol of the destination attribute, or if it is a global attribute). The only exception is the LHS synthesized attribute, which can only be computed when the rule has been completely parsed (in general). This attribute may depend on all global and non alterable local attributes in the whole rule.

Conditional expression lists form a special case. Since these lists do not start with an attribute name representing a destination attribute but with a guard expression, and there can in fact be multiple assignments in one list, it is not clear when to evaluate the list nor on which other attributes the guard and the other expressions may depend. To eliminate this problem, it is simply defined that a conditional expression list is evaluated as soon as the symbol whose number equals the current default symbol number is about to be parsed. This implies that both the guard and the assignments may only depend on global attributes and non alterable local attributes.
appearing before the current default symbol. Since at any time, only inherited attributes for the symbol that is currently being parsed are available, this also implies that all assignments in a conditional expression list must be made to attributes of the current default symbol.

The actual evaluation part of an assignment expression (the RHS) can have two formats: unconditional and conditional. The unconditional format syntax is equal to the standard pascal expression syntax. Expressions involve operators operating on attributes and constants of compatible types. Currently defined operators are: *, /, -, +, mod, div, >, <, <=, >=, or, and, not, shl, shr, after, succ, pred, bnot, band, bor, bxor, setb, clrb and tstb. If necessary, parentheses can be used to change the order of evaluation from the default one (pascal-like). All expressions are type checked, for operand-operand type compatibility (dyadic operators) and for operator-operand compatibility (all operators). If an incompatibility is detected, the compiler generates an appropriate error message. See table 3 for a complete list of all currently defined operators, their operand types and result types. For boolean compares, by definition the expression true > false evaluates to true.

The conditional format syntax is an if then else construct. Its exact form is:

<dest> := if <guard> then <expr1> else <expr2>

Here <dest> designates the attribute to which the result shall be assigned. Together with the current default symbol number, it uniquely defines the symbol for which this expression must be evaluated, and therefore on which attributes <guard>, <expr1> and <expr2> may depend. The <guard> expression must evaluate into a boolean result. If it evaluates to true, <expr1> is computed, otherwise <expr2> is evaluated. In either case, the result is assigned to <dest>. Note that <dest>, <expr1> and <expr2> must all be of compatible type (the compiler checks this).

There is one special system variable which can be used in expressions, that has not been mentioned yet. This variable is read-only and is called time. Its type is word, and its value represents the current system time, which means that it changes automatically every T time units. When the maximum value for a word type variable has been reached, the next increment will reset the value of time to 0.

Word compatible types are enumerated types, ranges and constants, as well as the simple types octet and word. Constants that are part of an enumerated type definition in the definitions section are of that enumerated type and not of type word,
even though they are mapped to numeric values. Two types are compatible when they are both boolean or when they are both word compatible, unless they are both enumerated types, in which case the types must be identical.

It should be noted that although there is a large similarity between the operators in the protocol grammar and those in the programming language pascal, there is one very important difference. All operators in the protocol grammar are functional operators, i.e. they operate like functions that return the result of their operation without affecting the operands on which they operate (without side effects). For example, the expression \( \text{setb}(4, p) \) will take the value of \( p \), set the fourth bit in it to 1 and return that new value as its result, without affecting \( p \) itself. The result must be assigned to some attribute or used as parameter to another function to form a valid expression construct. This allows for expressions such as:

\[
q := \text{setb}(a-3, p+1);
\]

\[
a := \text{clrb}(\text{setb}(9, b), \text{clrb}(4-a, c \text{ xor } 7));
\]
The productions section

<prod_section_body> → <production> [ <production> ] *
<production> → <prodrule> <prodcond> <prodeval>
<prodrule> → rule <prodsym> -> <prhs> [ || <inpevtname> ] *
<prhs> → @ | <prodsym> [ <prodsym> ] *
<prodsym> → <symname> <psattrlist>
<symname> → <name>
<inpevtname> → <name>
<prodccond> → ε | condition <expr>
<prodeval> → ε | evaluation <evallist>
<evallist> → <defexprlist> [ <defexprlist> ] *
<defexprlist> → <exprlist> | <symcode> <exprlist>
<symcode> → ( <number> ) :
<exprlist> → <partial_list> [ ; <partial_list> ] *
<partial_list> → <condeexprlist> | <assignexpr> | <loopeexpr>
<loopeexpr> → loop ( <init> if <expr> else <expr> ) <exprlist> endloop
<init> → ε | <exprlist>
<condeexprlist> → if <expr> then <exprlist> fi
<assignexpr> → <varid> := <expr>
<expr> → <condeexpr> | <uncondeexpr>
<condeexpr> → if <expr> then <exprlist> else <exprlist>
<uncondeexpr> → <simplex> <rexpr>
<simplex> → <expsign> <expterm> <rsimplex>
<rsimplex> → ε | <termopr> <expterm> <rsimplex>
<rexpr> → ε | <compopr> <simplex>
<expsign> → ε | + | –
<expterm> → <expfactor> <rexpterm>
<rexpterm> → ε | <dyaopr> <expfactor> <rexpterm>
<expfactor> → <varid> | <number> | true | false | time
 | ( <expr> ) | not <expfactor>
 | <monopr> ( <expr> )
 | <dyafunc> ( <expr> , <expr> )
<varid> → <name> <indices> [ . <name> <indices> ] *
<indices> → ε | [ ' ' <expr> '] ' <indices>
<dyaopr> → '*' | '/' | div | mod | and | band | bor | bxor
<termopr> → + | – | or
<compopr> → = | < | > | <= | >= | <=
<monopr> → shl | shr | after | succ | pred | ord | bnot
.dyafunc → setb | clrb | tstb
Example 15: rule

\[ Z (-r_1, +s_1) \rightarrow A (-t_1, +p_1) B (-t_1, +q_1) \]

classification

\[ r_1 \nless \ nmore \ 0 \]

evaluation

\[ s_1 := r_1 + p_1 * q_1 \]

(1): if \( r_1 = 1 \) then \( t_1 := r_1 \) fi

(2): \( t_1 := p_1 \)

defines a production rule that expands \( Z \) into the sequence \( AB \). Inherited attributes are \( r_1 \) (for \( Z \)) and \( t_1 \) (for both \( A \) and \( B \)). Synthesized attributes are \( s_1 \) (for \( Z \)), \( p_1 \) (for \( A \)) and \( q_1 \) (for \( B \)). The rule is only applicable when inherited attribute \( r_1 \) is not zero. When \( A \) is about to be parsed, the conditional expression is evaluated. When \( B \) is about to be parsed, the last evaluation expression is evaluated. After the rule has been completely parsed, \( s_1 \) will get the value as indicated by the first evaluation expression.

Tables 2 and 3 at the end of this chapter show lists of all keywords and operators currently defined in the protocol grammar language.

2 Operators, keywords and time

2.1 Operator precedence relations

Table 1 lists the operators and their priorities. These can also be derived from the BNF notation of the expression syntax, since priorities have been ‘hardwired’ in these metagrammar rules.

<table>
<thead>
<tr>
<th>Priority</th>
<th>Group</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>low</td>
<td>&lt;comoprt&gt;</td>
<td>= &lt; &gt; &lt;&gt; &lt;= &gt;=</td>
</tr>
<tr>
<td></td>
<td>&lt;expsign&gt;</td>
<td>- + (monadic)</td>
</tr>
<tr>
<td></td>
<td>&lt;termoprt&gt;</td>
<td>- + or (dyadic)</td>
</tr>
<tr>
<td></td>
<td>&lt;dyaprt&gt;</td>
<td>* / div mod and band bor bxor</td>
</tr>
<tr>
<td>high</td>
<td>&lt;monoprt&gt;, &lt;dyafunc&gt;</td>
<td>shl shr after succ pred ord bnot setb clrb tsetb</td>
</tr>
</tbody>
</table>

2.2 Value wrapping

When an assignment is made between two compatible but different type of variables, an automatic conversion is made from the source type to the destination type.
Such a conversion can change the value of the expression result. When assigning a word variable to an octet variable, all bits above the 8 lower source bits are discarded. Assigning a value to a range variable automatically forces the new value to be within the correct range using an auto wrapping mechanism. This also happens for enumerated type variables and range variables. If the destination range is \(<a, b>\) (including the boundaries) then a source value \(s\) will be wrapped to the real destination value \(d\) using the following formula:

\[
d := a + (s - a) \text{wrap} (b - a + 1)
\]

where:

\[
p \text{wrap} q = \begin{cases} 
  p & ; 0 \leq p < q \\
  q - 1 - (\text{bnor}(p) \mod q) & ; p < 0 \wedge q > 0 \\
  p \mod q & ; 0 < q \leq p 
\end{cases}
\]

Although this may seem a bit complex at first, it really is not. It is a generalization of the standard modulo arithmetic, where the lower limit is zero and the upper limit is a power of 2. Here, the upper limit may be any value (which is why a modulo operation is needed instead of simple bit-discarding or AND-ing) and the lower limit can also have any value (which means that the origin is shifted). In most cases however, the above formula can be simplified considerably. For example, for octet values (range \(<0, 255>\)) the formula becomes \(d := s \text{wrap} 256\), which in turn simplifies to \(d := s \text{band} 255\). During compilation every case is simplified as far as possible during code generation.

### 2.3 The concept of time

There are two system functions related to timing: \textit{time} and \textit{after}. This section describes how these functions relate to real time. In the following, the lowercase \(t\) represents the real time. First, a quantizer function \(Q\) is defined:

\[
Q : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N} \\
Q (k, T) = z & ; zT \leq k < (z+1)T
\]

Let \(\Delta\) be the minimum real time interval we wish to be able to distinguish and \(T_0\) be the moment in real time at which the system started operation. The quantized time since start up at time \(t\), called \(qtime\) is given by:

\[
qtime : \mathbb{N} \rightarrow \mathbb{N} \\
qtime (t) = Q (t - T_0 , \Delta)
\]
The function $qtime$ is the desired function, giving the number of expired intervals of size $\Delta$ since system start. However, no physical implementation can represent all natural numbers, since finite systems can only hold a finite set of numbers. In a digital system using $n$ bits for its values, the system time function $time$ is given by:

$$time : \emptyset \rightarrow \mathbb{N}$$

$$time = qtime(t) \mod 2^n$$

Figure 1. Circular Axis for time function

Such a system can only distinguish $2^n$ time intervals. This limitation can be represented graphically by a circular axis on which the $2^n$ intervals are marked equally spaced. When $time$ has reached the highest possible value it can hold, it will become zero in the next interval (see figure 1).

The result of the system function $after$ is also indicated in figure 1. The time domain is divided into two halves. At system time $\tau$, the function $after(\xi)$ will return true for those values of $\xi$ that belong to the first $2^{n-1}$ values immediately following $\tau$ and for $\tau$ itself. For all other values, it returns false. Mathematically, for an $n$ bit system:

$$after(\xi) = \begin{cases} 
true & ; (2^n + time - \xi) \mod 2^n < 2^{n-1} \\
false & ; (2^n + time - \xi) \mod 2^n \geq 2^{n-1} 
\end{cases}$$

This semantics for the after function implies that the maximum forward time span (for modelling time-outs) is $\Delta 2^{n-1}$. For a 16 bit system with $\Delta = 50$ msec., this results in a time span of approximately 1638 seconds, which is more than enough for most time out purposes.
### Table 2. Currently defined keywords and their use.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning and use</th>
</tr>
</thead>
<tbody>
<tr>
<td>grammar</td>
<td>Starts description of language in protocol grammar format</td>
</tr>
<tr>
<td>endgrammar</td>
<td>Ends description of language in protocol grammar format</td>
</tr>
<tr>
<td>definitions</td>
<td>Starts section containing type and constant definitions</td>
</tr>
<tr>
<td>terminals</td>
<td>Starts section containing declarations of terminal symbols</td>
</tr>
<tr>
<td>nonterminals</td>
<td>Starts section containing declarations of nonterminal symbols</td>
</tr>
<tr>
<td>formats</td>
<td>Starts section containing definitions of frame formats</td>
</tr>
<tr>
<td>attributes</td>
<td>Starts section containing declarations of attributes</td>
</tr>
<tr>
<td>channels</td>
<td>Starts section containing declarations of channels</td>
</tr>
<tr>
<td>events</td>
<td>Starts section containing declarations of event nonterminals</td>
</tr>
<tr>
<td>axiom</td>
<td>Starts section containing the name of the start symbol</td>
</tr>
<tr>
<td>productions</td>
<td>Starts section containing the language production rules</td>
</tr>
<tr>
<td>rule</td>
<td>Precedes the actual rule in a production rule subsection</td>
</tr>
<tr>
<td>condition</td>
<td>Precedes the condition expression of a rule</td>
</tr>
<tr>
<td>evaluation</td>
<td>Precedes the attribute evaluation expression section of a rule</td>
</tr>
<tr>
<td>discard</td>
<td>To specify the discard option of terminals and nonterminals</td>
</tr>
<tr>
<td>standard</td>
<td>To define a standard event</td>
</tr>
<tr>
<td>reset</td>
<td>To define a reset event</td>
</tr>
<tr>
<td>global</td>
<td>Used in attributes section to declare attribute(s) as global</td>
</tr>
<tr>
<td>local</td>
<td>Used in attributes section to declare attribute(s) as local</td>
</tr>
<tr>
<td>input</td>
<td>Used in channel declaration to declare input channel(s)</td>
</tr>
<tr>
<td>output</td>
<td>Used in channel declaration to declare output channel(s)</td>
</tr>
<tr>
<td>inherited</td>
<td>Used in attribute list of any symbol declaration to define mode of attribute as inherited</td>
</tr>
<tr>
<td>synthesized</td>
<td>Used in attribute list of any symbol declaration to define mode of attribute as synthesized</td>
</tr>
<tr>
<td>sdu</td>
<td>To reference data stored in main packet buffer memory (formats section)</td>
</tr>
<tr>
<td>pdu</td>
<td>To start definition of a frame format</td>
</tr>
<tr>
<td>management</td>
<td>To define a channel for inter processor communication</td>
</tr>
<tr>
<td>frames</td>
<td>To define a channel for packet I/O terminals</td>
</tr>
<tr>
<td>io_control</td>
<td>To define a channel for local hardware control</td>
</tr>
<tr>
<td>false</td>
<td>predefined boolean constant, given its usual meaning of false</td>
</tr>
<tr>
<td>true</td>
<td>predefined boolean constant, given its usual meaning of true</td>
</tr>
<tr>
<td>bool</td>
<td>predefined simple expression type, range = {false, true}</td>
</tr>
<tr>
<td>octets</td>
<td>predefined simple expression type, range = {0..255}</td>
</tr>
<tr>
<td>word</td>
<td>predefined simple expression type, range = {minint..maxint}</td>
</tr>
<tr>
<td>structure</td>
<td>Starts definition of a complex type in definitions section</td>
</tr>
</tbody>
</table>
Table 2 (continued).

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning and use</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>time</code></td>
<td>predefined read only system variable, representing system time in time units since start-up, modulo (maxint + 1)</td>
</tr>
<tr>
<td><code>for</code></td>
<td>Used in channel definition (separates direction from type)</td>
</tr>
<tr>
<td><code>if</code></td>
<td>Starts conditional expression list and RHS of conditional assignment</td>
</tr>
<tr>
<td><code>then</code></td>
<td>Used in conditional expression construct to separate guard expression from expression that is to be evaluated when guard evaluates to true.</td>
</tr>
<tr>
<td><code>else</code></td>
<td>Used in RHS of conditional assignment expressions to separate expression that is to be evaluated when guard evaluates to true from expression that is to be evaluated when guard evaluates to false. This latter one immediately follows the 'else' keyword.</td>
</tr>
<tr>
<td><code>fi</code></td>
<td>Used to mark the end of a conditional expression list.</td>
</tr>
<tr>
<td><code>loop</code></td>
<td>To start a loop definition in an evaluation section</td>
</tr>
<tr>
<td><code>endloop</code></td>
<td>To end a loop definition</td>
</tr>
</tbody>
</table>
### Table 3. The currently defined operators.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Operand &amp; Result Types</th>
<th>Syntax</th>
<th>Returned result</th>
</tr>
</thead>
<tbody>
<tr>
<td>div</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a \ div b )</td>
<td>integer division of ( a ) by ( b )</td>
</tr>
<tr>
<td>mod</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a \ mod b )</td>
<td>modulo of ( a ) by ( b )</td>
</tr>
<tr>
<td>band</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a \ band b )</td>
<td>bitwise AND of ( a ) and ( b )</td>
</tr>
<tr>
<td>bor</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a \ bor b )</td>
<td>bitwise OR of ( a ) and ( b )</td>
</tr>
<tr>
<td>bxor</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a \ bxor b )</td>
<td>bitwise excl. OR of ( a ) and ( b )</td>
</tr>
<tr>
<td>not</td>
<td>(bool) ( \Rightarrow ) bool</td>
<td>( \text{not} a )</td>
<td>boolean inversion of ( a )</td>
</tr>
<tr>
<td>or</td>
<td>(bool, bool) ( \Rightarrow ) bool</td>
<td>( a \ or b )</td>
<td>boolean OR of ( a ) and ( b )</td>
</tr>
<tr>
<td>and</td>
<td>(bool, bool) ( \Rightarrow ) bool</td>
<td>( a \ and b )</td>
<td>boolean AND of ( a ) and ( b )</td>
</tr>
<tr>
<td>bnot</td>
<td>(word) ( \Rightarrow ) word</td>
<td>( \text{bnot} (a) )</td>
<td>bitwise logical inversion</td>
</tr>
<tr>
<td>shl</td>
<td>(word) ( \Rightarrow ) word</td>
<td>( \text{shl} (a) )</td>
<td>( a ) shifted 1 bit to left</td>
</tr>
<tr>
<td>shr</td>
<td>(word) ( \Rightarrow ) word</td>
<td>( \text{shr} (a) )</td>
<td>( a ) shifted 1 bit to right</td>
</tr>
<tr>
<td>setb</td>
<td>(word, &lt;any&gt;) ( \Rightarrow ) &lt;any&gt;</td>
<td>( \text{setb} (a, b) )</td>
<td>( b ) with ( a^\text{th} ) bit set to 1</td>
</tr>
<tr>
<td>clr</td>
<td>(word, &lt;any&gt;) ( \Rightarrow ) &lt;any&gt;</td>
<td>( \text{clr} (a, b) )</td>
<td>( b ) with ( a^\text{th} ) bit set to 0</td>
</tr>
<tr>
<td>sstb</td>
<td>(word, &lt;any&gt;) ( \Rightarrow ) bool</td>
<td>( \text{sstb} (a, b) )</td>
<td>( a^\text{th} ) bit of ( b )</td>
</tr>
<tr>
<td>after</td>
<td>(word) ( \Rightarrow ) bool</td>
<td>( \text{after} (a) )</td>
<td>system time ( \text{after} ) ( a )</td>
</tr>
<tr>
<td>succ</td>
<td>(enum) ( \Rightarrow ) enum</td>
<td>( \text{succ} (a) )</td>
<td>next enumerated value</td>
</tr>
<tr>
<td>pred</td>
<td>(enum) ( \Rightarrow ) enum</td>
<td>( \text{pred} (a) )</td>
<td>previous enumerated value</td>
</tr>
<tr>
<td>ord</td>
<td>(&lt;n_str&gt;) ( \Rightarrow ) word</td>
<td>( \text{ord} (a) )</td>
<td>ordinal value of ( a )</td>
</tr>
<tr>
<td>+</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a + b )</td>
<td>addition of ( a ) and ( b )</td>
</tr>
<tr>
<td>-</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a - b )</td>
<td>subtraction of ( b ) from ( a )</td>
</tr>
<tr>
<td>*</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a * b )</td>
<td>multiplication of ( a ) by ( b )</td>
</tr>
<tr>
<td>/</td>
<td>(word, word) ( \Rightarrow ) word</td>
<td>( a / b )</td>
<td>division of ( a ) by ( b )</td>
</tr>
<tr>
<td>=</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a = b )</td>
<td>( a ) equals ( b )?</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a &lt;&gt; b )</td>
<td>( a ) does not equal ( b )?</td>
</tr>
<tr>
<td>&lt;</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a &lt; b )</td>
<td>( a ) less than ( b )?</td>
</tr>
<tr>
<td>&gt;</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a &gt; b )</td>
<td>( a ) greater than ( b )?</td>
</tr>
<tr>
<td>&lt;=</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a \leq b )</td>
<td>( a ) less than or equal to ( b )?</td>
</tr>
<tr>
<td>&gt;=</td>
<td>(&lt;n_str&gt;, &lt;n_str&gt;) ( \Rightarrow ) bool</td>
<td>( a \geq b )</td>
<td>( a ) greater than or equal to ( b )?</td>
</tr>
</tbody>
</table>

Other recognized special characters are: \( ( ) : = , ; @ . [ ] | \)

\(<\text{n\_str}>\) == any type which is not a structure or an array, \(<\text{any}>\) == any type
3 The compiler job language

In the previous sections the language for actual protocol grammar definitions has been defined. However, a protocol is usually split in a number of parallel executable parts which are each described in a separate protocol grammar and which all need to be compiled (see [Bloks93b], chapter 7). This could be done by invoking the compiler for each grammar, but some of the information concerning the structure of the entire protocol engine can be derived by examining all grammars involved. Also, different parts might need compilation with different options and it should be easy to recompile an entire protocol after a change in an external parameter. To accomplish this, the compiler operates by executing jobs.

A job file is a description (text) file containing a number of commands for the compiler to execute. The compiler works by first parsing this entire file, storing the commands it encounters and then executing them. Currently, there are only 3 possible commands: \texttt{linkfile}, \texttt{configuration} and \texttt{compile}.

\begin{verbatim}
<job_body>  →  linkfile <filename>
  | configuration <filename>
  | compile <filename> [ <switch> ] *

<switch>  →  -e <filename>
  | -l <filename>
  | -c
  | -h <directoryname>
  | -d
  | -s
  | -v <number>
  | -g <filename>
  | -n
  | -f <filename>
\end{verbatim}

The file and directory names used in these commands must be valid for the operating system of the computer that runs the compiler. Therefore, job files are usually system dependent, in contrast to the ProGrIL files themselves. A name can be enclosed in single quotes (') if it contains space characters (some operating systems allow spaces in file names). Commands can be distributed over multiple lines and are case insensitive. As with grammar descriptions, it is possible to insert comment text between double quotes wherever a space-character would also be allowed. To avoid
confusion in the large number of files involved in the compilation of an entire protocol into a protocol engine, a naming convention for files is introduced. They are listed in Table 4. Note that these conventions are not enforced by the compiler. Only the '.hex' extension for code output files is hardcoded in the compiler.

**Table 4. File naming conventions.**

<table>
<thead>
<tr>
<th>Extension</th>
<th>Contents</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>.job</td>
<td>job commands</td>
<td>user specified input</td>
</tr>
<tr>
<td>.cfg</td>
<td>configuration parameters</td>
<td>optional user specified input</td>
</tr>
<tr>
<td>.grm</td>
<td>ProGrIL description</td>
<td>user specified input</td>
</tr>
<tr>
<td>.log</td>
<td>error report</td>
<td>compiler generated on error</td>
</tr>
<tr>
<td>.lst</td>
<td>output listing</td>
<td>compiler generated on request</td>
</tr>
<tr>
<td>.hex</td>
<td>code in Intel Hex format</td>
<td>compiler generated on request</td>
</tr>
<tr>
<td>.lnk</td>
<td>link file</td>
<td>compiler generated on request</td>
</tr>
<tr>
<td>.tst</td>
<td>reverse compile test file</td>
<td>compiler generated/debug</td>
</tr>
</tbody>
</table>

The *linkfile* command specifies the name of the output file that contains all information for the hardware linker concerning the structure (interconnections, message formats, etc.) of the protocol engine. There is only one such file for an entire job, containing the information for all compiled grammar files. If there are more than one linkfile commands in a job file, only the last one counts. If no linkfile command is given, no such file will be generated. The format of the data in the link file depends on the requirements of the hardware compiler, which has not yet been constructed. It is therefore subject to change and will be defined in a later stage.

The *configuration* command is used to specify the name of the global parameter file that the compiler may use. The structure of the configuration file is described later. It contains values for a number of parameters such as bus widths, memory depths, etc. of the parts of the grammar processors. Because the compiler implements the code to run on these processors, it must know these values. Alternatively, the compiler can derive these values from the ProGrIL descriptions, in which case the values from the configuration file can be considered as maximum allowed values (and should be set accordingly). The compiler will generate error messages if any protocol grammar cannot be implemented on a grammar processor whose parameters are as given in the configuration file. The configuration command can be omitted, in which case the compiler uses built-in defaults.
The *compile* command is used to actually perform the compilation of a ProGrIL description. The first argument of this command is the name of a grammar file in the format explained in section 1. This argument may optionally be followed by a set of one-character compile switches (which may also be uppercase) and in some cases a file or directory name, as listed in table 5 below.

**Table 5. Compiler options.**

<table>
<thead>
<tr>
<th>Switch</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>-e &lt;filename&gt;</code></td>
<td>specifies name of error log output file, which is generated only when an error or warning is issued by the compiler. If omitted, the filename is derived from the grammar filename by stripping its extension and concatenating the result with <code>.LOG</code>.</td>
</tr>
<tr>
<td><code>-l &lt;filename&gt;</code></td>
<td>specifies the name of the list file (always generated) containing a full listing of information derived from the grammar and/or generated by the compiler itself.</td>
</tr>
<tr>
<td><code>-c</code></td>
<td>code generation should be attempted.</td>
</tr>
<tr>
<td><code>-h &lt;directoryname&gt;</code></td>
<td>code generation should be attempted, and if successful, the resulting code should be written to <code>.HEX</code> files in the directory with the given name (generates about 28 files).</td>
</tr>
<tr>
<td><code>-d</code></td>
<td>listing should go to the screen (not to list file). If both l and d are given, the one given last takes effect.</td>
</tr>
<tr>
<td><code>-s</code></td>
<td>include set data in output listing (results of first, last, follow and lookahead set computation).</td>
</tr>
<tr>
<td><code>-v &lt;number&gt;</code></td>
<td>set verbose level from 0 to 5, specifying how much information is output in list file (mostly concerning generated code): 0: generate everything 1: skip bits (ascii representation of binary output vectors) 2: skip bits and messages 3: skip bits, messages and original expressions 4: skip everything during code generation 5: skip everything (including data).</td>
</tr>
<tr>
<td><code>-g &lt;filename&gt;</code></td>
<td>specifies different configuration file to be used for this compile command only.</td>
</tr>
<tr>
<td><code>-n</code></td>
<td>output nonterminal derivation graph and detected cycles in the list file.</td>
</tr>
<tr>
<td><code>-f &lt;filename&gt;</code></td>
<td>specifies name of packet format information listing file. This file contains some information about packet formats and how they are constructed from data sent to packet assemblers resp. received from packet disassemblers.</td>
</tr>
</tbody>
</table>

Switches can be specified in any order. When two switches conflict, the later one takes effect. To turn an option on, you simply put the switch in the compile command in the job file.
4 Configuration files

Configuration files contain information about the widths and depths of various parts of the grammar processors, such as buses, memories and the number of available event inputs. These values override the defaults in the compiler. Not all values need to be specified in a configuration file; only those that are different from the defaults are required. Because the settings are specified in textual format, they can be easily edited with any text editor and they can be specified in any order. As usual, the compiler is case insensitive while parsing the configuration file. The values that are specified must be 'reasonable' before they are accepted by the compiler. This means that there is an upper limit for all values. When at least one value is too large, an error message is generated and job execution is aborted.

Table 6. Configuration settings.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Max</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>data</td>
<td>16</td>
<td>32</td>
<td>system word width: all code is compiled for a system with this data word width</td>
</tr>
<tr>
<td>codeaddress</td>
<td>11</td>
<td>16</td>
<td>size of code rom addresses (bits)</td>
</tr>
<tr>
<td>action</td>
<td>7</td>
<td>10</td>
<td>size of action procedure bus (bits)</td>
</tr>
<tr>
<td>symbol</td>
<td>8</td>
<td>10</td>
<td>size of symbol (token) bus (bits)</td>
</tr>
<tr>
<td>ruleaddress</td>
<td>16</td>
<td>16</td>
<td>size of rule rom addresses (bits)</td>
</tr>
<tr>
<td>nrs</td>
<td>12</td>
<td>12</td>
<td>size of NRS indices (bits)</td>
</tr>
<tr>
<td>laset</td>
<td>64</td>
<td>1024</td>
<td>number of elements in LA set</td>
</tr>
<tr>
<td>alufield</td>
<td>5</td>
<td>8</td>
<td>size of alu function field</td>
</tr>
<tr>
<td>srcfield</td>
<td>3</td>
<td>8</td>
<td>size of source operand fields</td>
</tr>
<tr>
<td>dstfield</td>
<td>4</td>
<td>8</td>
<td>size of destination operand field</td>
</tr>
<tr>
<td>datafield</td>
<td>16</td>
<td>32</td>
<td>size of data field</td>
</tr>
<tr>
<td>offsetfield</td>
<td>16</td>
<td>32</td>
<td>size of offset field</td>
</tr>
<tr>
<td>rwfield</td>
<td>3</td>
<td>8</td>
<td>size of read/write control field</td>
</tr>
<tr>
<td>flowfield</td>
<td>4</td>
<td>8</td>
<td>size of flow control field</td>
</tr>
<tr>
<td>maxinchannels</td>
<td>2</td>
<td>8</td>
<td>max. number of input channels</td>
</tr>
<tr>
<td>maxstandardevents</td>
<td>4</td>
<td>16</td>
<td>max. number of standard events</td>
</tr>
<tr>
<td>maxinputevents</td>
<td>4</td>
<td>16</td>
<td>max. number of input events</td>
</tr>
<tr>
<td>maxresetevents</td>
<td>4</td>
<td>16</td>
<td>max. number of reset events</td>
</tr>
</tbody>
</table>
The format of the settings in the configuration file is very simple:

\[
\text{<config\_body>} \rightarrow [ \text{<setting>} = \text{<number>} ] *
\]

\[
\text{<setting>} \rightarrow \text{data} \\
\text{codeaddress} \\
\text{action} \\
\text{symbol} \\
\text{ruleaddress} \\
\text{nrs} \\
\text{laset} \\
\text{alufield} \\
\text{srcfield} \\
\text{dstfield} \\
\text{datafield} \\
\text{offsetfield} \\
\text{rwfield} \\
\text{flowfield} \\
\text{maxinchannels} \\
\text{maxstandardevents} \\
\text{maxinputevents} \\
\text{maxresetevents}
\]

The maximum values are partly imposed by limitations of the pascal compiler used to construct the ProGrIL compiler (such as the 32 bit data word size limit) and partly chosen arbitrarily (such as the maximum number of events). The compiler will work correctly with parameter values up to the maximum values. A change in the maximum values might require a change/rewrite of some parts of the compiler.
Bibliography

Aho72 Aho, A.V. and J.D. Ullman

Aho86 Aho, A.V. et al.
Compilers: Principles, techniques and tools.
Amsterdam: Addison Wesley, 1986.

Anderson85a Anderson, D.P. and L.H. Landweber
Protocol specification by real-time attribute grammars.

Anderson85b Anderson, D.P. and L.H. Landweber
A grammar-based methodology for protocol specification and implementation.

Bloks91 Bloks, R.H.J.
A protocol engine architecture.

Bloks92 Bloks, R.H.J.
A metagrammar for ProGrIL.
Internal Report no. PRO/EB/92022.

Bloks93a Bloks, R.H.J.
Code generation for the attribute evaluator of the protocol engine grammar processor unit.

Bloks93b Bloks, R.H.J.
A grammar based approach towards the automatic implementation of data communication protocols in hardware.
Dissertation, Eindhoven University of Technology.
To be published approx. May 1993.

Bochmann80 Bochmann, G.V.
A general transition model for protocols and communication services.

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.

Denning78 Denning, P.J. et al.
Machines, languages and computation.
Gelli87  Gelli, P.  
Evaluation and comparison of three specification languages: SDL, Lotos and Estelle.  

Haas85  Haas, O.  
Spezifikation von Kommunikationsprotokollen auf der Basis attribuierter Grammatiken. (in German).  

Haas86  Haas, O.  
Formal protocol specification based on attribute grammars.  

Harangozo678  Harangozó, J.  
Protocol definition with formal grammars.  

Kain72  Kain, R.Y.  
Automata theory: Machines and languages.  

Knuth68  Knuth, D.E.  
Semantics of context-free languages.  

Lewis81  Lewis, H.R. and C.H. Papadimitriou  
Elements of the theory of computation.  

Salomaa73  Salomaa, A.  
Formal languages. (from: ACM Monograph Series)  

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.

Sunshine82  Sunshine, C.A.  
Formal modeling of communication protocols.  
In: Computer networks and simulation II. Ed. by S. Schoemaker.  

Venkatraman86  Venkatraman, R.C. and T.F. Piatkowski  
A formal comparison of formal protocol specification techniques.  
(246) Maagt, P.J.J. de
A SYNTHESIS METHOD FOR COMBINED OPTIMIZATION OF MULTIPLE ANTENNA PARAMETERS AND ANTENNA PATTERN STRUCTURE.

(247) Jonkman, L. and T. Spangova Kwantel
DECOMPOSITIONAL STATE ASSIGNMENT WITH REUSE OF STANDARD DESIGNS: Using counters as sub-machines and using the method of minimal adjacencies to select the state chains and the state codes

(248) Bierwagens, M.J. and J.M. Vliegheuwers
DERIVATION AND VERIFICATION OF A MODEL OF THE SYNCHRONOUS MACHINE WITH RECTIFIER WITH TWO DAMPER WINDINGS ON THE DIRECT AXIS

(249) Zhu, Y.C. and A.C.P.M. Backx, P. Eyhoff
MULTIVARIABLE PROCESS IDENTIFICATION FOR ROBUST CONTROL.

(250) Platjehofer, F.M. and P.J.M. Cluitmans, H.M. Kuipers
ERMAS: Design and formal specification of a datamodel for a clinical research database system

THE ASCIS DATA FLOW GRAPH: Semantics and textual format.

(252) Cheg, J. and P.J.J. de Maagt, M.H.A.J. Herbon

(253) Heerd, S.W.H. de
A PWM CURRENT-SOURCE INVERTER FOR INTERCONNECTION BETWEEN A PHOTOVOLTAIC ARRAY AND THE UTILITY LINE.

(254) Veld, M. van de and P.J.M. Cluitmans
ECC ANALYSIS FOR MONITORING OF ANESTHETIC DEPTH.

(255) Smolders, A.B.
AN EFFICIENT METHOD FOR ANALYZING MICROSTRIP ANTENNAS WITH A DIELECTRIC COVER USING A SPECTRAL DOMAIN MOMENT METHOD

(256) Backx, A.C.P.M. and A.A.H. Damen
IDENTIFICATION FOR THE CONTROL OF MIMO INDUSTRIAL PROCESSES.

(257) Maagt, P.J.J. de and H G ter Morsche, J.J.M. van den Broek
A SPATIAL RECONSTRUCTION TECHNIQUE APPLICABLE TO MICROWAVE RADIPORTY.
(258) Vreeswijkers, J. M.
DERIVATION OF A MODEL OF THE EXCITER OF A BRUSHLESS SYNCHRONOUS MACHINE

(259) Orlov, V. R.
EFFECT MOTION AS THE ORIGIN OF THE 1/F CONDUCTANCE NOISE IN SOLIDS

(260) Rooijackers, J. E.
ALGORITHMS FOR SPEECH CODING SYSTEMS BASED ON LINEAR PREDICTION.

(261) Reem, T. J. J. van den and A. H. M. Damen, Martin Klopstra
IDENTIFICATION FOR ROBUST CONTROL USING AN H- INFINITY NORM.

(262) Grooten, M. and W. van Etten
LASER LINewidth MEASUREMENT IN THE PRESENCE OF RIN AND USING THE RECTICULAR\nHETERODYNE METHOD

(263) Snolders, A. B.
RIGOROUS ANALYSIS OF THICK MICROSTRIP ANTENNAS AND WIRE ANTENNAS EMBEDDED IN A SUBSTRATE.

(264) Freiks, L. W. and P. J. M. Cluitmans. J. M. van Gils
THE ADAPTIVE RESONANCE THEORY NETWORK: (Clustering-1 behaviour in relation with brainstem\nauditory evoked potential patterns.

MANUFACTURING AND CHARACTERIZATION OF GaAs/AlGaAs MULTIPLE QUANTUMWELL RIDGE \nWAVEGUIDE LASERS.

(266) Cluitmans, J. M.
USING GENETIC ALGORITHMS FOR SCHEDULING DATA FLOW GRAPHS.

(267) Jozwik, L. and A. P. H. van Dijk
A METHOD FOR GENERAL SIMULTANEOUS FULL DECOMPOSITION OF SEQUENTIAL MACHINES\nAlgorithms and implementation.

(268) Reem, H. van den and W. van Etten, M. H. C. de Krom, P. van Beekhoven, P. Rooijackers,\nL. Niessen, F. de Lelijer
AN OPTICAL ASK AND FSK PHASE DIVERSITY TRANSMISSION SYSTEM.

(269) Putten, P. H. A. van der
MULTIDISCIPLINARY SPECIFICEREN EN ONTWERPEN VAN MICROELEKTRONICA IN PRODUCENTEN (in Dutch).