SDL PARSER

PROJECT WORK DONE AT P-653

KUMARAGURU COLLEGE OF TECHNOLOGY

PROJECT REPORT

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE AWARD OF THE DEGREE OF

MASTER OF ENGINEERING

OF BHARATHIAR UNIVERSITY , COIMBATORE.

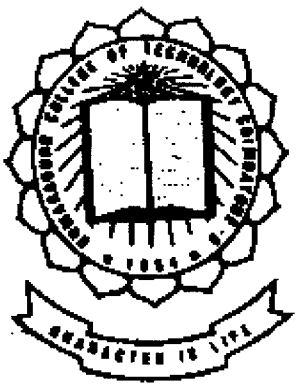
SUBMITTED BY

N. CHITRA DEVI

Reg. No. 0037K0003

GUIDED BY

Mr. K.R. BHASKAR M.S.



Department of Computer Science & Engineering

KUMARAGURU COLLEGE OF TECHNOLOGY

Coimbatore – 641 006

Department of Computer Science & Engineering

Kumaraguru College Of Technology

(Affiliated to the Bharathiar University)

Coimbatore 641 006

CERTIFICATE

This is to certify that the project work entitled

SDL PARSER

Done by

N. Chitra devi

Reg. No. 0037K0003

Submitted in partial fulfillment of the requirements for the award of the degree of

Master of Engineering of Bharathiar University.

S. Jh 20/12/01

Professor and Head

18/12

Internal Guide

Submitted for University Examination held on.....20/12/2001.....

20/12

Internal Examinar

External Examiner

DECLARATION

I here by declare that the project entitled '**SDL PARSER**' submitted to **Bharathiar University** as the project work of Master of Engineering (COMPUTER SCIENCE AND ENGINEERING) Degree, is a record of original work done my me under the supervision and guidance of **Mr. K.R. BASKAR M.S.,** Assistant professor, Computer Science and Engineering Department, Kumaraguru College of Technology and this project work has not found the basis for the award of any Degree / Diploma / Associate ship / Fellowship or similar title to any candidate of any university.

Place :..Coimbatore.....

Date :..20.12.2001.....

Signature of the Student

Countersigned by

(Internal Guide)

CONTENTS

ACKNOWLEDGEMENT

I take this opportunity to express my respectful note of thanks to **Prof. K. Arumugam, B.E., (Hons), M.S. (U.S.A), M.I.E.,** Correspondent, Kumaraguru College of Technology for giving me a chance to do M.E. Computer Science And Engineering in his esteemed institution.

I wish to express my sincere gratitude to **Dr. K. K. Padmanabhan, B,Sc. (Engg)., M.Tech., Ph.D.,** esteemed Principal, Kumaraguru College of Technology for providing me the necessary facilities in the college.

I wish to express my heartfelt thanks and a deep sense of gratitude to **Prof.S.Thangaswamy, B.E.(Hons), Ph.D.,** The Head Of Department Of Computer Science And Engineering who motivated me by giving valuable ideas and suggestions.

I wish my sincere thanks to my Guide cum Course Coordinator **Mr. K.R. Bhaskaran M.S.,** Assistant Professor, Department Of Computer Science And Engineering without whose motivation and guidance I would not have been able to embark on a project of this magnitude.

Last but not the least, I thank my beloved family members, friends, department teaching and non teaching staffs who have been a pillar of support right from the start, until the completion of the project.

SYNOPSIS

SDL – 2000 is a Specification and Description Language standardized as ITU (International Telecommunication Union) Recommended Z.100. SDL has been designed to specify and design the functional behaviour of telecommunication systems. It has been used in a variety of fields, including data communication protocol specification.

The main objective of the project is to develop a parser for SDL-2000. In addition, the software provides other features like generation of AST (Abstract Syntax Tree), 'help' facility for software usage, provision to get version information etc. It has been developed using a compiler– compiler tool , called ANTLR (Another Tool For Language Recognition) version. 2.7.1.

ANTLR, ANother Tool for Language Recognition is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions. The reasons for choosing ANTLR are it supports syntactic, semantic predicates , object oriented concepts and arbitrary lookahead .

SDL parser is a semantic analyzer which consumes the system specification and description in phrase notation (SDL/PR) , analyzes the same and provides system display for errors , if any . This analysis is done based upon the SDL lexical rules and grammar definition specified in the grammar files.

The parser generated is a predicated – LL[k] parser based on top-down approach. The grammar rule definitions for lexer , parser and tree parser are defined separately in grammar files based on the EBNF (Extended Backus Naur Form) notation supported in ANTLR. A main module to link lexer , parser and tree parser is defined . In this module,the input is buffered and the lexer is instantiated with the buffer. After lexer initialization it is binded with parser by passing the lexer as an

argument to it. The vocabulary generated by lexer is exported in lexer and imported in parser and they are made to share a common namespace hence the tokens generated by lexer are accessible in parser. The tree parser is made to handle tree transformations by augmenting it with parser.

1. INTRODUCTION

1.1 CURRENT STATUS OF THE PROBLEM

SDL is a **Specification and Description Language** standardized as **ITU** (International Tele communication Union) Recommendation **Z.100.** SDL has been designed to specify and design the functional behaviour of Tele communication system. It has been used in a variety of fields, including data communication protocol specification. The SDL standard is updated every four years since 1976. The previous updation was done in 1996 and the latest was done in 2000.

In this project an attempt was made to develop a syntax recognizer for Specification and Description Language – 2000 (SDL-2000). The SDL tool developed, does the lexical and syntactical analysis. In addition it takes care of symbol table management and error handling. The Compiler supports other features like it provides version information, help facility for knowing compiler usage. It allows user to have a AST tree structure of the parsed tree.

1.2 RELEVANCE AND IMPORTANCE OF THE TOPIC

Scope

The purpose of recommending SDL (Specification and Description Language) is to provide a language for unambiguous specification and description of the behaviour of telecommunications systems.

The terms specification and description are used with the following meaning:

a) a specification of a system is the description of its required behaviour; and
b) a description of a system is the description of its actual behaviour; that is its implementation.

Objective

The general objectives for recommending SDL is that :

a) is easy to learn, use and interpret;
b) provides unambiguous specification for ordering, tendering and design, while
   also allowing some issues to be left open;
c) may be extended to cover new developments;
d) is able to support several methodologies of system specification and design.
e) the ability to be used as a wide spectrum language from requirements to implementation ;
f) suitability for real-time, stimulus-response systems;
g) presentation in a graphical form;
h) a model based on communicating processes (extended finite state machines)
i) object oriented description of SDL components

Application

The main area of application for SDL is the specification of the behaviour of aspects of real time systems, and the design of such systems. Applications in the field of telecommunications include:

a) call and connection processing (for example, call handling, telephony signaling,
   metering) in switching systems;
b) maintenance and fault treatment (for example alarms, automatic fault clearance,
   routine tests) in general telecommunications systems;
c) system control (for example, overload control, modification and extension procedures);
d) operation and maintenance functions, network management;

e) data communication protocols;

f) telecommunications services.

SDL can, of course, be used for the functional specification of the behaviour of any object whose behaviour can be specified using a discrete model; that is where the object communicates with its environment by discrete messages.

SDL is a rich language and can be used for both high level informal (and/or formally incomplete) specifications, semi-formal and detailed specifications. The user must choose the appropriate parts of SDL for the intended level of communication and the environment in which the language is being used. Depending on the environment in which a specification is used, then many aspects may be left to the common understanding between the source and the destination of the specification.

Thus SDL may be used for producing:

a) facility requirements;

b) system specifications;

c) ITU-T Recommendations, or other similar Standards (international, regional or national);

d) system design specifications;

e) detailed specifications;

f) system design descriptions (both high level and detailed enough to directly produce implementations);

g) system testing descriptions (in particular in combination with MSC and TTCN).

The user organization can choose the appropriate level of application of SDL.

2. LITERATURE SURVEY

2.1 VERSIONS OF SDL

The Z.100 Recommendation is updated every four year by the ITU, since 1976. The latest updates occurred in

- 1992: major release
- 1996: minor modifications. Only an Addendum to Z.100 document has been released,
- 2000: major release

2.2 ITU – T RECOMMENDATION PUBLICATIONS FOR SDL 2000

- Z.100 (11/99) Specification and Description Language (SDL)
- Z.105 (11/99) SDL combined with ASN.1 modules;
- Z.107 (11/99) SDL with embedded ASN.1;
- Z.109 (11/99) SDL combined with UML.

2.3 DIFFERENCES BETWEEN SDL-92 AND SDL-2000

A strategic decision was made to keep SDL stable for the period 1992 to 1996, so that at the end of this period only a limited number of changes were made to SDL. These were published as Addendum 1 to Z.100 (10/96) rather than updating the SDL-92 document. Although this version of SDL was sometimes called SDL-96, it was small change compared with the change from SDL-88 to SDL-92. The changes were:

a) harmonizing signals with remote procedures and remote variables;

b) harmonizing channels and signal routes;

c) adding external procedures and operations;

d) allowing a block or process to be used as a system;

e) state expressions;

f) allowing packages on blocks and processes;

g) parameterless operators.

These have now been incorporated into Z.100, together with a number of other changes to produce a version of SDL known as SDL-2000. In this Recommendation the language defined by Z.100 (03/93) with Addendum 1 to Z.100 (10/96) is still called SDL-92.

The advantages of language stability, which was maintained over the period from 1992 to 1996, began to be outweighed by the need to update SDL to support and better match other languages that are frequently used in combination with SDL. Also modern tools and techniques have made it practical to generate software more directly from SDL specifications, but further significant gains could be made by incorporating better support for this use in SDL. While SDL-2000 is largely an upgrade of SDL-92, it was agreed that some incompatibility with SDL-92 was justified; otherwise the resulting language would have been too large, too complex and too inconsistent. This subclause provides information about the changes.

Changes have been made in a number of areas, which focus on simplification of the language, and adjustment to new application areas:

a) adjustment of syntactical conventions to other languages with which SDL is used;
b) harmonization of the concepts of system, block and process to be based on "agent" , and merging of the concept of signal route into the concept channel;
c) interface descriptions;
d) exception handling;
e) support for textual notation of algorithms within SDL/GR;
f) composite states;
g) replacement of the service construct with the state aggregation construct;

h) new model for data;

j) constructs to support the use of ASN.1 with SDL previously in Z.105 (03/95).

Other changes are: nested packages, direct containment of blocks and processes in blocks, **out**-only parameters.

On the syntactic level, SDL-2000 is case-sensitive. Keywords are available in two spellings: all uppercase or all lowercase. Keywords of SDL-2000 that are not keywords of SDL-92 are:

abstract, aggregation, association, break, choice, composition, continue, endexceptionhandler, endmethod, endobject, endvalue, exception, exceptionhandler, handle, method, object, onexception, ordered, private, protected, public, raise, value.

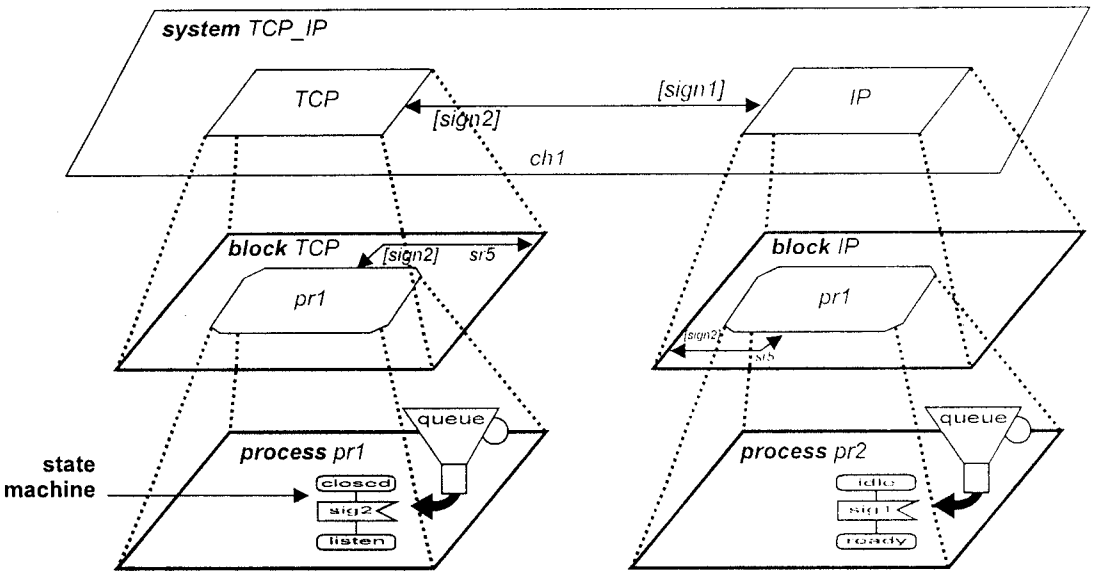
The following keywords of SDL-92 are not keywords in SDL-2000:

all, axioms, constant, endgenerator, endnewtype, endrefinement, endservice, error, fpar, generator, imported, literal, map, newtype, noequal, ordering, refinement, returns, reveal, reverse, service, signalroute, view, viewed.

A small number of constructs of SDL-92 are not available in SDL-2000: view expression, generators, block substructures, channel substructures, signal refinement, axiomatic definition of data, macro diagrams. These constructs were rarely (if ever) used, and the overhead of keeping them in the language and tools did not justify their retention.

2.4 OVERVIEW OF SDL

In SDL 2000, the architecture is modeled as a system containing blocks, as depicted in figure. Each block may contain either blocks or processes. Each process contains an extended finite state machine. State machines communicate by exchanging signals through channels (or signal routes).



Signal (sig1 or sig2 in Figure) arriving on a state machine are queued. By consuming a signal from its queue, a state machine executes a transition from one state to another state. During the execution of a transition, a wide range of actions can be performed by a state machine: signal transmission to another state machine, assignment, procedure or operator call, loop, process instance creation etc.

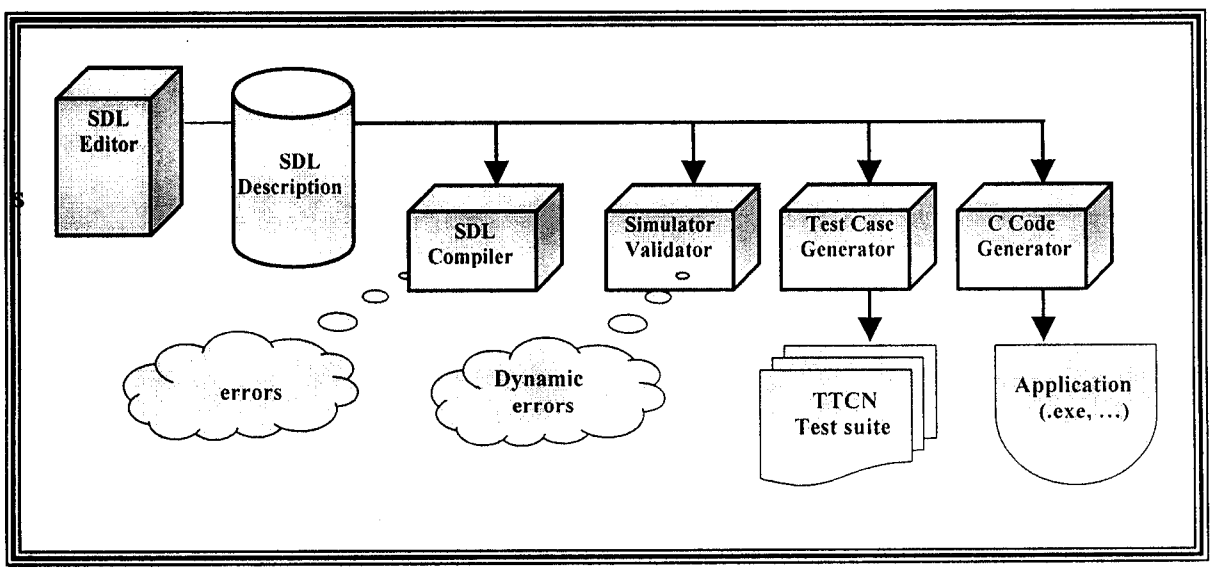
As opposed to several other notations, the execution semantics of SDL is accurately described, including the semantics of actions in state machines transitions.

The communication by signals between SDL state machines is asynchronous: after transmitting a signal, the state machine continues its execution, it does not wait. A state machine may also call a remote (i.e. defined in another state machine)

procedure, in which case it is blocked waiting for the remote procedure return. A signal can carry parameters.

Data type are described using predefined types or constructs such as Integer, Boolean, Character, struct, Array, String, Charstring, Powerset. A subset of ASN.1 can be used to describe more complex data types, using constructs such as choice (similar to union in C), optional fields, Bistring or Octetstring. Timers can be used for example to avoid waiting forever a response to an event. To ease configuration management and team work, an SDL description can be split into packages. SDL is object-oriented: it provides the notions of classes, inheritance, polymorphism (in SDL 2000), etc. found in object-oriented programming languages.

SDL is a Formal Description Technique (FDT), like Estelle and LOTOS for example. But SDL is more used in the software industry and benefits of more tool support than Estelle and LOTOS. SDL is frequently used with MSCs (Message Sequence Charts)



Life of an SDL description

2.5 DEFINITIONS

The are numerous terms defined throughout the Z.100 Recommendation and therefore only a few key terms are given in this clause.

2.5.1 agent:

The term agent is used to denote a system, block or process that contains one or more extended finite state machines.

2.5.2 block:

A block is an agent that contains one or more concurrent blocks or processes and may also contain an extended finite state machine that owns and handles data within the block.

2.5.3 body:

A body is a state machine graph of an agent, procedure, composite state, or operation.

2.5.4 channel:

A channel is a communication path between agents.

2.5.5 environment:

The environment of the system is everything in the surroundings that communicates with the system in an SDL-like way.

2.5.6 gate:

A gate represents a connection point for communication with an agent type, and when the type is instantiated it determines the connection of the agent instance with other instances.

2.5.7 instance:

An instance is an object created when a type is instantiated.

2.5.8 object:

The term object is used for data items that are references to values.

2.5.9 pid:

The term pid is used for data items that are references to agents.

2.5.10 procedure:

A procedure is an encapsulation of part of the behaviour of an agent, that is defined in one place but may be called from several places within the agent. Other agents can call a remote procedure.

2.5.11 process:

A process is an agent that contains an extended finite state machine, and may contain other processes.

2.5.12 signal:

The primary means of communication is by signals that are output by the sending
agent and input by the receiving agent.

2.5.13 sort:

A sort is a set of data items that have common properties.

2.5.14 state:

An extended finite state machine of an agent is in a state if it is waiting for a stimulus.

2.5.15 stimulus:

A stimulus is an event that can cause an agent that is in a state to enter a transition.

2.5.16 system:

A system is the outermost agent that communicates with the environment.

2.5.17 timer:

A timer is an object owned by an agent that causes a timer signal stimulus to occur at a specified time.

```
<letter> ::=
                    <uppercase letter> | <lowercase letter>
<uppercase letter> ::=
                        A | B | C | D | E | F | G | H | I | J | K | L | M
                        | N | O | P | Q | R | S | T | U | V | W | X | Y | Z
<lowercase letter> ::=
                        a | b | c | d | e | f | g | h | i | j | k | l | m
                        | n | o | p | q | r | s | t | u | v | w | x | y | z
<decimal digit> ::=
                        0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<quoted operation name> ::=
                    <quotation mark> <infix operation name> <quotation mark>
                  | <quotation mark> <monadic operation name> <quotation mark>
<infix operation name> ::=
                    or  |  xor  |  and  |  in  |  mod  |  rem
                  | <plus sign>                | <hyphen>
                  | <asterisk>                 | <solidus>
                  | <equals sign>              | <not equals sign>
                  | <greater than sign>        | <less than sign>
                  | <less than or equals sign> | <greater than or equals sign>
                  | <concatenation sign>       | <implies sign>
<monadic operation name> ::=
                    <hyphen> | not
<character string> ::=
                    <apostrophe> { <general text character>
                  | <special>
                  | <apostrophe> <apostrophe>
                  } * <apostrophe>
<apostrophe> <apostrophe> represents an <apostrophe> within a <character string>.

<hex string> ::=
                    <apostrophe> { <decimal digit>
                    | a | b | c | d | e | f
                    | A | B | C | D | E | F
                    }* <apostrophe> { H | h }
<bit string> ::=
                    <apostrophe> { 0  | 1
                    }* <apostrophe> { B | b }
<note> ::=
                    <solidus> * <asterisk> * note text * <asterisk> / <solidus>
<note text> ::=
                    {       <general text character>
                    |       <other special>
                    |       <asterisk>+ <not asterisk or solidus>
                    |       <solidus>
                    |       <apostrophe> }*
<not asterisk or solidus> ::=
                    <general text character> | <other special> | <apostrophe>
<text> ::=
                    { <general text character> | <special> | <apostrophe> }*
<general text character> ::=
                    <alphanumeric> | <other character> | <space>
<composite special> ::=
                    |       <result sign>
                    |       <composite begin sign>
                    |       <composite end sign>
                    |       <concatenation sign>
```

| <history dash sign>
| <greater than or equals sign>
| <implies sign>
| <is assigned sign>
| <less than or equals sign>
| <not equals sign>
| <qualifier begin sign>
| <qualifier end sign>

<result sign> ::=
<hyphen> <greater than sign>

<composite begin sign> ::=
<left parenthesis> <full stop>

<composite end sign> ::=
<full stop> <right parenthesis>

<concatenation sign> ::=
<solidus> <solidus>

<history dash sign> ::=
<hyphen> <asterisk>

<greater than or equals sign> ::=
<greater than sign> <equals sign>

<implies sign> ::=
<equals sign> <greater than sign>

<is assigned sign> ::=
<colon> <equals sign>

<less than or equals sign> ::=
<less than sign> <equals sign>

<not equals sign> ::=
<solidus> <equals sign>

<qualifier begin sign> ::=
<less than sign> <less than sign>

<qualifier end sign> ::=
<greater than sign> <greater than sign>

<special> ::=
<solidus> | <asterisk> | <other special>

<other special> ::=
<exclamation mark> | <number sign>
| <left parenthesis> | <right parenthesis>
| <plus sign> | <comma> | <hyphen>
| <full stop> | <colon> | <semicolon>
| <less than sign> | <equals sign> | <greater than sign>
| <left square bracket> | <right square bracket>
| <left curly bracket> | <right curly bracket>

<other character> ::=
<quotation mark> | <dollar sign> | <percent sign>
| <ampersand> | <question mark> | <commercial at>
| <reverse solidus> | <circumflex accent> | <underline>
| <grave accent> | <vertical line> | <tilde>

<exclamation mark> ::= !
<quotation mark> ::= "
<left parenthesis> ::= (
<right parenthesis> ::=)
<asterisk> ::= *
<plus sign> ::= +
<comma> ::= ,

<hyphen> ::= -
<full stop> ::= .
<solidus> ::= /
<colon> ::= :
<semicolon> ::= ;
<less than sign> ::= <
<equals sign> ::= =
<greater than sign> ::= >
<left square bracket> ::= [
<right square bracket> ::=]
<left curly bracket> ::= {
<right curly bracket> ::= }
<number sign> ::= #
<dollar sign> ::= $
<percent sign> ::= %
<ampersand> ::= &
<apostrophe> ::= '
<question mark> ::= ?
<commercial at> ::= @
<reverse solidus> ::= \
<circumflex accent> ::= ^
<underline> ::= _
<grave accent> ::= `
<vertical line> ::= |
<tilde> ::= ~
<keyword>::=

abstract	**active**	**adding**
aggregation	**alternative**	**and**
any	**as**	**association**
atleast	**block**	**break**
call	**channel**	**choice**
comment	**composition**	**connect**
connection	**constants**	**continue**
create	**dcl**	**decision**
default	**else**	**endalternative**
endblock	**endchannel**	**endconnection**
enddecision	**endexceptionhandler**	**endinterface**
endmacro	**endmethod**	**endobject**
endoperator	**endpackage**	**endprocedure**
endprocess	**endselect**	**endstate**
endsubstructure	**endsyntype**	**endsystem**
endtype	**endvalue**	**env**
exception	**exceptionhandler**	**export**
exported	**external**	**fi**
finalized	**for**	**from**
gate	**handle**	**if**
import	**in**	**inherits**
input	**interface**	**join**
literals	**macro**	**macrodefinition**
macroid	**method**	**methods**
mod	**nameclass**	**nextstate**
nodelay	**none**	**not**
now	**object**	**offspring**
onexception	**operator**	**operators**
optional	**or**	**ordered**
out	**output**	**package**
parent	**priority**	**private**

	procedure	**protected**	**process**
	provided	**public**	**raise**
	redefined	**referenced**	**rem**
	remote	**reset**	**return**
	save	**select**	**self**
	sender	**set**	
	signal	**signallist**	**signalset**
	size	**spelling**	**start**
	state	**stop**	**struct**
	substructure	**synonym**	**syntype**
	system	**task**	**then**
	this	**timer**	**to**
	try	**type**	**use**
	value	**via**	**virtual**
	with	**xor**	

<space> ::=

2.6.2 Parser rules

2.6.2.1 Organization of SDL specification

Framework

An <sdl specification> can be described as a monolithic <system specification> or as a collection of <package>s and <referenced definition>s. A <package> allows definitions to be used in different contexts by "using" the package in these contexts, that is, in systems or packages which may be independent. A <referenced definition> is a definition that has been removed from its defining context to gain overview within one system description.

Concrete grammar

<sdl specification> ::=
[<specification area>]
{ <package> | <system specification> } <package>* <referenced definition>*

<system specification> ::=
<textual system specification> | <graphical system specification>

<package> ::=
<package definition> | <package diagram>

Concrete textual grammar

<textual system specification> ::=
<agent definition>
| <textual typebased agent definition>

Package

In order for a type definition to be used in different systems it has to be defined as part of a *package*. Definitions as parts of a package define types, signal lists, remote specifications and synonyms. Definitions within a package are made visible to another scope unit by a package use clause.

Concrete textual grammar

<package definition> ::=
{<package use clause>}*
<package heading> <end>
{<entity in package>}*
endpackage [<package name>] <end>

<package heading> ::=
package [<qualifier>] <package name>
[<package interface>]

<entity in package> ::=
<agent type definition>
| <agent type reference>
| <package definition>
| <package reference>
| <signal definition>
| <signal reference>
| <signal list definition>
| <remote variable definition>
| <data definition>
| <data type reference>
| <procedure definition>
| <procedure reference>
| <remote procedure definition>
| <composite state type definition>
| <composite state type reference>
| <exception definition>
| <select definition>
| <macro definition>
| <interface reference>
| <association>

<package reference> ::=
package [<qualifier>] <package name> **referenced** <end>

<package use clause> ::=
use <package identifier> [/ <definition selection list>] <end>

<definition selection list> ::=
<definition selection> { , <definition selection>}*

<definition selection> ::=
[<selected entity kind>] <name>

<selected entity kind> ::=
system type
| **block type**
| **process type**
| **package**
| **signal**
| **procedure**

```
                    |   remote procedure
                    |   type
                    |   signallist
                    |   state type
                    |   synonym
                    |   remote
                    |   exception
                    |   interface
<package interface> ::=
                        public <definition selection list>
```

Referenced definition

Concrete grammar

```
<referenced definition> ::=
                        <definition> | <diagram>
```

Concrete textual grammar

```
<definition> ::=
                        <package definition>
                    |   <agent definition>
                    |   <agent type definition>
                    |   <composite state>
                    |   <composite state type definition>
                    |   <procedure definition>
                    |   <operation definition>
                    |   <macro definition>
```

2.6.2.2 Structural type definitions

Agent types

An agent type is a system, block or process type. When the type is used to define an agent, the agent is of corresponding kind (system, block or process).

Concrete textual grammar

```
<agent type definition> ::=
                    <system type definition> | <block type definition> | <process type definition>
<agent type structure> ::=
                    [<valid input signal set>]
                    {   {   <entity in agent>
                    |       <channel definition>
                    |       <gate in definition>
                    |       <agent definition>
                    |       <agent reference>
                    |       <textual typebased agent definition> }x
                    [   <state partitioning>]
                    |   {   <entity in agent>
```

```
                    |       <gate in definition> }^x
                      <agent type body> }
<type preamble> ::=
                    [   <virtuality> | <abstract> ]
<agent type additional heading> ::=
                    [<formal context parameters>] [<virtuality constraint>]
                    <agent additional heading>
<agent type reference> ::=
                      <system type reference>
                    |   <block type reference>
                    |   <process type reference>
<agent type body> ::=
                    [[<on exception>] <start> ]
                    { <state> | <exception handler> | <free action> } ^x
```

System type

A system type definition is a top-level agent type definition. It is denoted by the keyword **system type**. A system type definition must not be contained in any other agent or agent type definition. A system type can neither be abstract nor virtual.

Concrete textual grammar

```
<system type definition> ::=
                      <package use clause>*
                      <system type heading> <end> <agent type structure>
                      endsystem type [ [<qualifier>] <system type name>] <end>
<system type heading> ::=
                      system type [<qualifier>] <system type name>
                      <agent type additional heading>
<system type reference> ::=
                      system type <system t ype identifier> <type reference properties>
```

Block type

Concrete textual grammar

```
<block type definition> ::=
                      <package use clause>*
                      <block type heading> <end> <agent type structure>
                      endblock type [ [<qualifier>] <block type name> ] <end>
<block type heading> ::=
                      <type preamble>
                      block type [<qualifier>] <block type name>
                          <agent type additional heading>
<block type reference> ::=
                      <type preamble>
                      block type <block type identifier> <type reference properties>
```

Process type

Concrete textual grammar

```
<process type definition> ::=
                        <package use clause>*
                        <process type heading> <end> <agent type structure>
                        endprocess type [ [<qualifier>] <process type name>] <end>
<process type heading> ::=
                        <type preamble>
                        process type [<qualifier>] <process type name>
                        <agent type additional heading>
<process type reference> ::=
                        <type preamble>
                        process type <process type identifier> <type reference properties>
```

Composite state type

Concrete textual grammar

```
<composite state type definition> ::=
            {<package use clause>}*
            <composite state type heading> <end> substructure
                    [<valid input signal set>]
                    {<gate in definition>}*
                    {<state connection points>}*
                    {<entity in composite state>}*
                    <composite state body>
            endsubstructure state type [ [ <qualifier> ] <composite state type name> ]
        <end>
        |   {<package use clause>}*
            <state aggregation type heading> <end> substructure
                    [<valid input signal set>]
                    {<gate in definition>}*
                    {<state connection points>}*
                    {<entity in composite state>}*
                    <state aggregation body>
            endsubstructure state type [ [ <qualifier> ] <composite state type name> ]
        <end>
<composite state type heading> ::=
            [<virtuality>]
            state type [ <qualifier> ] <composite state type name>
                    [<formal context parameters>] [<virtuality constraint>]
                    [<specialization>]
                    [<agent formal parameters>]
<state aggregation type heading> ::=
            [<virtuality>]
            state aggregation type [ <qualifier> ] <composite state type name>
                    [<formal context parameters>] [<virtuality constraint>]
                    [<specialization>]
                    [<agent formal parameters>]
<composite state type reference> ::=
            <type preamble>
```

state type *<composite state type identifier> <type reference properties>*

Type expression

A type expression is used for defining one type in terms of another.

Concrete textual grammar

```
<type expression> ::=
                        <base type> [<actual context parameters>]
<base type> ::=
                        <identifier>
```

Definitions based on types

A typebased agent definition defines an agent instance set according to a type denoted by <type expression>. The defined entities get the properties of the types that they are based on.

Concrete textual grammar

```
<textual typebased agent definition> ::=
                        <textual typebased system definition>
                    |   <textual typebased block definition>
                    |   <textual typebased process definition>
```

System definition based on system type

Concrete textual grammar

```
<textual typebased system definition> ::=
                        <typebased system heading> <end>
<typebased system heading> ::=
                        system <system name> <colon> <system type expression>
```

Block definition based on block type

Concrete textual grammar

```
<textual typebased block definition> ::=
                        block <typebased block heading> <end>
<typebased block heading> ::=
                        <block name> [<number of instances>] <colon> <block
                            type expression>
```

Process definition based on process type

Concrete textual grammar

<textual typebased process definition> ::=
process <typebased process heading> <end>
<typebased process heading> ::=
<process name> [<number of instances>] <colon> <process type expression>

Composite state definition based on composite state type

Concrete textual grammar

<textual typebased state partition definition> ::=
state aggregation <typebased state partition heading>
<end>
<typebased composite state> ::=
<state name> [<actual parameters>] <colon> <composite state type expression>

Abstract type

Concrete grammar

<abstract> ::=
abstract

Gate

Gates are defined in agent types (block types, process types) or state types and represent connection points for channels, connecting instances of these with other instances or with the enclosing frame symbol.

Concrete textual grammar

<gate in definition> ::=
<textual gate definition> | <textual interface gate definition>
<textual gate definition> ::=
gate <gate> [**adding**] <gate constraint> [<gate constraint>] <end>
<gate> ::=
<gate name>
<gate constraint> ::=
{ **out** [**to** <textual endpoint constraint>] | **in** [**from** <textual endpoint constraint>] }
[**with** <signal list>]
<textual endpoint constraint> ::=
[**atleast**] <identifier>

<textual interface gate definition> ::=
gate { **in** | **out** } **with** <interface identifier> **end**

Context parameters

In order for a type definition to be used in different contexts, both within the same system specification and within different system specifications, types may be parameterized by context parameters. Context parameters are replaced by actual context parameters. The following type definitions can have formal context parameters: system type, block type, process type, procedure, signal, composite state, interface and data type.

Context parameters can be given constraints, that is, required properties any entity denoted by the corresponding actual identifier must have. The context parameters have these properties inside the type.

Concrete textual grammar

<formal context parameters> ::=
<context parameters start> <formal context parameter list> <context parameters end>
<formal context parameter list> ::=
<formal context parameter> {<end> <formal context parameter> }*
<actual context parameters> ::=
<context parameters start> <actual context parameter list> <context parameters end>
<actual context parameter list> ::=
[<actual context parameter>] , [<actual context parameter>] }*
<actual context parameter> ::=
<identifier> | <constant primary>
<context parameters start> ::=
<less than sign>
<context parameters end> ::=
<greater than sign>
<formal context parameter> ::=
<agent type context parameter>
| <agent context parameter>
| <procedure context parameter>
| <remote procedure context parameter>
| <signal context parameter>
| <variable context parameter>
| <remote variable context parameter>
| <timer context parameter>
| <synonym context parameter>
| <sort context parameter>
| <exception context parameter>
| <composite state type context parameter>
| <gate context parameter>
| <interface context parameter>

Agent type context parameter

Concrete textual grammar

<agent type context parameter> ::=
process type | **block type** <agent type name> [<agent type constraint>]
<agent type constraint> ::=
atleast <agent identifier> | <agent signature>

Agent context parameter

Concrete textual grammar

<agent context parameter> ::=
{ **process** | **block** } <agent name> [<agent constraint>]
<agent constraint> ::=
{ **atleast** | <colon> } <agent type identifier> | <agent signature>
<agent signature> ::=
<sort list>

Procedure context parameter

Concrete textual grammar

<procedure context parameter> ::=
procedure <procedure name> <procedure constraint>
<procedure constraint> ::=
atleast <procedure identifier> | <procedure signature in constraint>
<procedure signature in constraint> ::=
[(<formal parameter> { , <formal parameter> }*)] [<result>]

Remote procedure context parameter

Concrete textual grammar

<remote procedure context parameter> ::=
remote procedure <procedure name> <procedure signature in constraint>

Signal context parameter

Concrete textual grammar

<signal context parameter> ::=
signal <signal name> [<signal constraint>]
{ , <signal name> [<signal constraint>] }*

<signal constraint> ::=
atleast <signal identifier> | <signal signature>
<signal signature> ::=
<sort list>

Variable context parameter

Concrete textual grammar

<variable context parameter> ::=
dcl <variable name> { , <variable name>}* <sort>
{ , <variable name> { , <variable name>}* <sort> }*

Remote variable context parameter

Concrete textual grammar

<remote variable context parameter> ::=
remote <remote variable name> { , <remote variable name>}* <sort>
{ , <remote variable name> { , <remote variable name>}* <sort>
}*

Timer context parameter

Concrete textual grammar

<timer context parameter> ::=
timer <timer name> [<timer constraint>]
{ , <timer name> [<timer constraint>] }*
<timer constraint> ::=
<sort list>

Synonym context parameter

Concrete textual grammar

<synonym context parameter> ::=
synonym <synonym name> <synonym constraint>
{, <synonym name> <synonym constraint> }*
<synonym constraint> ::=
<sort>

Sort context parameter

Concrete textual grammar

<sort context parameter> ::=
[{ **value** | **object** }] **type** <sort name> [<sort constraint>]
<sort constraint> ::=
atleast <sort> | <sort signature>
<sort signature> ::=
literals <literal signature> { , <literal signature> }*
[**operators** <operation signature in constraint> { , <operation signature in constraint> }*
]
[**methods** <operation signature in constraint> { , <operation signature in constraint> }* |
| **operators** <operation signature in constraint> { , <operation signature in constraint> }*
[**methods** <operation signature in constraint> { , <operation signature in constraint> }* |
| **methods** <operation signature in constraint> { , <operation signature in constraint> }*
<operation signature in constraint> ::=
<operation name>[(<formal parameter> { , <formal parameter> }*)] [<result>]
| <name class operation> [<result>]

Exception context parameter

Concrete textual grammar

<exception context parameter>::=
exception <exception name> [<exception constraint>]
{ , <exception name> [<exception constraint>] }*
<exception constraint>::=
<sort list>

Composite state type context parameter

Concrete textual grammar

<composite state type context parameter> ::=
state type <composite state type name> | <composite state type constraint> |
<composite state type constraint> ::=
atleast <composite state type identifier> | <composite state type signature>
<composite state type signature> ::=
<sort list>

Gate context parameter

Concrete textual grammar

<gate context parameter> ::=
gate <gate> <gate constraint> [<gate constraint>]

Interface context parameter

Concrete textual grammar

```
<interface context parameter> ::=
                    interface <interface name> [<interface constraint>]
                        { , <interface name> [<interface constraint>] }*
<interface constraint> ::=
                    atleast <interface identifier>
```

Sepcilization

A type may be defined as a specialization of another type (the supertype), yielding a new subtype. A subtype may have properties in addition to the properties of the supertype, and it may redefine virtual local types and transitions.

Adding properties

Concrete textual grammar

```
<specialization> ::=
                    inherits <type expression> [adding]
```

Virtual type

An agent type, procedure or state type may be specified as a virtual type when it is defined locally to another type (denoted as the *enclosing* type). A virtual type may be redefined in specializations of the enclosing type.

Concrete textual grammar

```
<virtuality> ::=
                    virtual | redefined | finalized
<virtuality constraint> ::=
                    atleast <identifier>
```

Associations

An association expresses a binary relationship between two entity types.

Concrete textual grammar

```
<association> ::=
                                    association [<association name>] <association kind> <end>
```

<association kind>::=
<association not bound kind>
| <association end bound kind>
| <association two ends bound kind>
| <composition not bound kind>
| <composition part end bound kind>
| <composition composite end bound kind>
| <composition two ends bound kind>
| <aggregation not bound kind>
| <aggregation part end bound kind>
| <aggregation aggregate end bound kind>
| <aggregation two ends bound kind>

<association not bound kind> ::=
from <association end> **from** <association end>

<association end bound kind>::=
from <association end> **to** <association end>

<association two ends bound kind>::=
to <association end> **to** <association end>

<composition not bound kind>::=
from <association end> **composition** <association end>

<composition part end bound kind>::=
to <association end> **composition** <association end>

<composition composite end bound kind> ::=
from <association end> **to composition** <association end>

<composition two ends bound kind>::=
to <association end> **to composition** <association end>

<aggregation not bound kind>::=
from <association end> **aggregation** <association end>

<aggregation part end bound kind>::=
to <association end> **aggregation** <association end>

<aggregation aggregate end bound kind>::=
from <association end> **to aggregation** <association end>

<aggregation two ends bound kind>::=
to <association end> **to aggregation** <association end>

<association end> ::=
[<visibility>] [**as** <role name>] <linked type identifier> [**size** <multiplicity>] [**ordered**]

<linked type identifier> ::=
<agent type identifier>
| <data type identifier>
| <interface identifier>

<multiplicity> ::=
<range condition>

Agents

An agent definition defines an (arbitrarily large) set of agents. An agent is characterized by having variables, procedures, a state machine (given by an explicit or implicit composite state type) and sets of contained agents.

There are two kinds of agents: *blocks* and *processes*. A *system* is the outermost block. The state machine of a block is interpreted *concurrently* with its contained

agents, while the state machine of a process is interpreted *alternating* with its contained agents.

Concrete textual grammar

```
<agent definition> ::=
                    <system definition> | <block definition> | <process definition>
<agent structure> ::=
                    [<valid input signal set>]
                    {   {   <entity in agent>
                        |   <channel to channel connection>
                        |   <channel definition>
                        |   <gate in definition>
                        |   <agent definition>
                        |   <agent reference>
                        |   <textual typebased agent definition> }*
                        [<state partitioning>]
                    |   {   <entity in agent>
                        |   <gate in definition> }*
                        <agent body>
```

The <state partitioning> must have the same name as the containing agent. It defines the state machine of the agent. If <agent structure> can be understood both as <state partitioning> and <agent body>, it is interpreted as <agent body>.

```
<agent instantiation> ::=
                    [   <number of instances>]
                    <agent additional heading>
<agent additional heading> ::=
                    [<specialization>] [<agent formal parameters>]
<entity in agent> ::=
                    <signal definition>
                |   <signal reference>
                |   <signal list definition>
                |   <variable definition>
                |   <remote procedure definition>
                |   <remote variable definition>
                |   <data definition>
                |   <data type reference>
                |   <timer definition>
                |   <interface reference>
                |   <macro definition>
                |   <exception definition>
                |   <procedure reference>
                |   <procedure definition>
                |   <composite state type definition>
                |   <composite state type reference>
                |   <select definition>
                |   <agent type definition>
                |   <agent type reference>
                |   <association>
<agent reference> ::=
                    <block reference>
```

```
        |       <process reference>
<valid input signal set> ::=
                signalset [<signal list>] <end>

<agent body> ::=
                <state machine graph>
<state machine graph>::=
                [<on exception>] <start>
                { <state> | <exception handler> | <free action> } *
<agent formal parameters> ::=
                ( <parameters of sort> , <parameters of sort>}* )
<parameters of sort> ::=
                <variable name> , <variable name>}* <sort>
<number of instances> ::=
                ( [<initial number>] [ , [<maximum number>] ] )
<initial number> ::=
                <Natural simple expression>
<maximum number> ::=
                <Natural simple expression>
```

System

A System is the outermost agent and has the *Agent-kind* **SYSTEM**. It is defined by a <system definition> or a <system diagram>. The semantics of agents applies with the additions provided in this subclause.

Concrete textual grammar

```
<system definition> ::=
                {<package use clause>}*
                <system heading> <end> <agent structure>
                endsystem [<system name>] <end>
<system heading>::=
                system <system name> <agent additional heading>
```

Block

A block is an agent with the *Agent-kind* **BLOCK**. The semantics of agents therefore applies with the additions provided in this subclause. A block is defined by a <block definition> or a <block diagram>.

Concrete textual grammar

```
<block definition> ::=
                {<package use clause>}*
                <block heading> <end> <agent structure>
```

```
                    endblock [ [<qualifier>] <block name> ] <end>
<block heading> ::=
                    block [<qualifier>] <block name> <agent instantiation>
<block reference> ::
                    block <block name> [<number of instances>] referenced <end>
```

Process

A process is an agent with the *Agent-kind* **PROCESS**. The semantics of agents therefore applies with the additions provided in this subclause.

Concrete textual grammar

```
<process definition> ::=
                    {<package use clause>}*
                    <process heading> <end> <agent structure>
                    endprocess [ [<qualifier>] <process name> ] <end>
<process heading> ::
                    process [<qualifier>] <process name> <agent instantiation>
<process reference> ::=
                    process <process name> [<number of instances>] referenced <end>
```

Procedure

Procedures are defined by means of procedure definitions. The procedure is invoked by means of a procedure call identifying the procedure definition. Parameters are associated with a procedure call.

Concrete textual grammar

```
<procedure definition> ::=
                    <external procedure definition>
                |   {<package use clause>}*
                    <procedure heading> <end>
                            <entity in procedure>*
                            [<procedure body>]
                    endprocedure [ [<qualifier>] <procedure name> ] <end>
                |   {<package use clause>}*
                    <procedure heading>
                            [ <end> <entity in procedure>+ ]
                            [<virtuality>] <left curly bracket>
                                    <statement list>
                            <right curly bracket>
<procedure preamble> ::=
                    <type preamble> [ exported [ as <remote procedure identifier> ]]
<procedure heading> ::=
                    <procedure preamble>
                    procedure [<qualifier>] <procedure name>
                    [<formal context parameters>] [<virtuality constraint>]
```

[<specialization>]
[<procedure formal parameters>]
[<procedure result>] [<raises>]

<procedure formal parameters> ::=
(<formal variable parameters> , <formal variable parameters> }*)

<formal variable parameters> ::=
<parameter kind> <parameters of sort>

<parameter kind> ::=
[**in/out** | **in** | **out**]

<procedure result> ::=
<result sign> [<variable name>] <sort>

<raises> ::=
raise <exception identifier> , <exception identifier>}*

<entity in procedure> ::=
<variable definition>
| <data definition>
| <data type reference>
| <procedure reference>
| <procedure definition>
| <composite state type definition>
| <composite state type reference>
| <exception definition>
| <select definition>
| <macro definition>

<procedure signature> ::=
[(<formal parameter> { , <formal parameter> }*)] [<result>] [<raises>]

<procedure body> ::=
[<on exception>] [<start>] { <state> | <exception handler> | <free action> } *

<external procedure definition> ::=
procedure <procedure name> <procedure signature> **external** <end>

<procedure reference> ::=
<type preamble>
procedure <procedure identifier> <type reference properties>

Communication

Channel

A channel is allowed to connect the two directions of a bidirectional gate to each other.

Concrete textual grammar

<channel definition> ::=
channel [<channel name>] [**nodelay**]
<channel path> [<channel path>]
endchannel [<channel name>] <end>

<channel path> ::=
from <channel endpoint>
to <channel endpoint> [**with** <signal list>] <end>

<channel endpoint> ::=
{ <agent identifier> | <state identifier> | **env** | **this** }[<via gate>]

<via gate> ::=
via <gate>

Connection

Concrete textual grammar

```
<channel to channel connection> ::=
                    connect <external channel identifiers>
                    and <channel identifiers> <end>
<external channel identifiers> ::=
                    <channel identifier> { , <channel identifier>}*
<channel identifiers> ::=
                    <channel identifier> { , <channel identifier>}*
```

Signal

Concrete textual grammar

```
<signal definition>::=
                    <type preamble>
                    signal <signal definition item> { , <signal definition item> }* <end>
<signal definition item> ::=
                    <signal name>
                    [<formal context parameters>]
                    [<virtuality constraint>]
                    [<specialization>]
                    [<sort list>]
<sort list> ::=
                    ( <sort> { , <sort>}* )
<signal reference> ::=
                    <type preamble>
                    signal <signal identifier> <type reference properties>
```

Signal list definition

A <signal list identifier> may be used in <signal list> as shorthand for a list of signal identifiers, remote procedures, remote variables, timer signals, and interfaces.

Concrete textual grammar

```
<signal list definition> ::=
                    signallist <signal list name> <equals sign> <signal list> <end>
<signal list> ::=
                    <signal list item> { , <signal list item>} *
<signal list item> ::=
                    <signal identifier>
                |   ( <signal list identifier> )
                |   <timer identifier>
                |   [ procedure ] <remote procedure identifier>
                |   [ interface ] <interface identifier>
                |   [ remote ] <remote variable identifier>
```

Remote procedures

A client agent may call a procedure defined in another agent by a request to the server agent through a remote procedure call of a procedure in the server agent.

Concrete textual grammar

<remote procedure definition> ::=
remote procedure <remote procedure name> [**nodelay**]
<procedure signature> <end>
<remote procedure call> ::=
call <remote procedure call body>
<remote procedure call body> ::=
<remote procedure identifier> [<actual parameters>]
<communication constraints>
<communication constraints> ::=
to <destination> | **timer** <timer identifier> | <via path>}*

Remote variables

In SDL, a variable is always owned by, and local to, an agent instance. Normally the variable is visible only to the agent instance that owns it and to the contained agents. If an agent instance in another agent needs to access the data items associated with a variable, a signal interchange with the agent instance owning the variable is needed.

This can be achieved by the following shorthand notation, called imported and exported variables. The shorthand notation may also be used to export data items to other agent instances within the same agent.

Concrete textual grammar

<remote variable definition> ::=
remote <remote variable name> ,<remote variable name>}* <sort> [**nodelay**]
<remote variable name> , <remote variable name>}* <sort> [**nodelay**]}*
<end>
<import expression> ::=
import (<remote variable identifier> <communication constraints>)
<export> ::=
export (<variable identifier> { , <variable identifier> }*)

Behaviour

Start

Concrete textual grammar

```
<start> ::=
        start [<virtuality>] [<state entry point name>] <end> [<on exception>]
            <transition>
```

State

Concrete textual grammar

```
<state> ::=
        <basic state> | <composite state application>
```

Basic State

Concrete textual grammar

```
<basic state> ::=
        state <state list> <end> [<on exception>]
            {       <input part>
            |       <priority input>
            |       <save part>
            |       <spontaneous transition>
            |       <continuous signal> } *
        [ endstate [<state name>] <end> ]
<state list> ::=
        <state name> { , <state name> }*
        | <asterisk state list>
<asterisk state list> ::=
        <asterisk> [ ( <state name> { , <state name>}* ) ]
```

Composite state application

A <composite state application> describes that the state machine has a composite state. The properties of the composite state are defined either as part of the <composite state application>, by a referenced composite state, or by a composite state type.

Concrete textual grammar

```
<composite state application> ::=
            state <composite state list> <end> [<on exception>]
                {       <input part>
```

```
                |   <priority input>
                |   <save part>
                |   <spontaneous transition>
                |   <continuous signal>
                |   <connect part> }*
[ endstate [ <state name> ] <end> ]
```

Input

Concrete textual grammar

```
<input part> ::=
                    input [<virtuality>] <input list> <end>
                    [<on exception>] [<enabling condition>] <transition>
<input list> ::=
                    <stimulus> { , <stimulus> } *
                    | <asterisk input list>
<stimulus> ::=
                    <signal list item>
                    [ ( [ <variable> ] { , [ <variable> ] }* ) | <remote procedure reject> ]
<remote procedure reject> ::=
                    raise <exception raise>
<asterisk input list> ::=
                    <asterisk>
```

Priority Input

In some cases, it is convenient to express that reception of a signal takes priority over reception of other signals. This can be expressed by means of priority input.

Concrete textual grammar

```
<priority input> ::=
                    priority input [<virtuality>]
                    <priority input list> <end> [<on exception>]<transition>
<priority input list> ::=
                    <stimulus> , <stimulus>}*__ [ <variable> ] }*
```

Continuous signal

In describing systems, the situation may arise where a transition should be interpreted when a certain condition is fulfilled. A continuous signal interprets a

Boolean expression and the associated transition is interpreted when the expression returns the predefined Boolean value true.

Concrete textual grammar

```
<continuous signal> ::
                        provided [<virtuality>]
                                <continuous expression> <end>
                        [ priority <priority name> <end> ] [<on exception>] <transition>
<continuous expression> ::=
                        <Boolean expression>
<priority name> ::=
                        <Natural literal name>
```

Enabling condition

An enabling condition makes it possible to impose an additional condition on the consumption of a signal, beyond its reception as well as on a spontaneous transition.

Concrete textual grammar

```
<enabling condition> ::=
                        provided <provided expression> <end>
<provided expression> ::=
                        <Boolean expression>
```

Save

A save specifies a set of signal identifiers and remote procedure identifiers whose instances are not relevant to the agent in the state to which the save is attached, and which need to be saved for future processing.

Concrete textual grammar

```
<save part> ::=
                        save [<virtuality>] <save list> <end>
<save list> ::=
                        <signal list>
                    |   <asterisk save list>
<asterisk save list> ::=
                        <asterisk>
```

Spontaneous transition

A spontaneous transition specifies a state transition without any signal reception.

Concrete textual grammar

```
<spontaneous transition> ::=
                        input [<virtuality>] <spontaneous designator> <end>
                        [<on exception>] [<enabling condition>] <transition>
<spontaneous designator> ::=
                        none
```

Label

Concrete textual grammar

```
<label> ::
                        <connector name> :
<free action> ::=
                        connection
                            <transition>
                        [ endconnection [ <connector name> ] <end> ]
```

State machine and Composite state

A composite state is a state that may either consist of sequentially interpreted substates (with associated transitions), or of an aggregation of substates interpreted in an interleaving mode. A substate is a state, so a substate may in turn be a composite state.

Concrete textual grammar

```
<composite state> ::=
                        <composite state graph> | <state aggregation>
```

Composite State Graph

In a composite state graph, the transitions are interpreted sequentially.

Concrete textual grammar

```
<composite state graph> ::=
                        {<package use clause>}*
                        <composite state heading> <end> substructure
                                [<valid input signal set>]
                                {<gate in definition>}*
                                <state connection points>*
```

```
                        <entity in composite state>*
                        <composite state body>
                    endsubstructure [ [<qualifier>] <composite state name> ] <end>
<composite state heading> ::
                    state [<qualifier>] <composite state name>
                    [<agent formal parameters>]
                    <entity in composite state> ::=
<variable definition>
                |   <data definition>
                |   <select definition>
                |   <data type reference>
                |   <macro definition>
                |   <procedure definition>
                |   <exception definition>
                |   <composite state type definition>
                |   <composite state type reference>
<composite state body> ::=
                    [<on exception>] <start>* { <state> | <exception handler> | <free action> }*
<composite state reference> ::=
                    state substructure <composite state name> referenced <end>
```

State aggregation

A state aggregation is a partitioning of a composite state.

Concrete textual grammar

```
                    <state aggregation> ::=
                    {<package use clause>}*
                        <state aggregation heading> <end> substructure
                        <state connection points>*
                        <entity in composite state>*
                        <state aggregation body>
                    endsubstructure [ [<qualifier>] <composite state name> ] <end>
<state aggregation heading> ::=
                    state aggregation [<qualifier>] <composite state name>
                    [<agent formal parameters>]
<state aggregation body> ::=
                    { <state partitioning>
                    | <state partition connection> }+
<state partitioning> ::=
                    <textual typebased state partition definition>
                    | <composite state reference>
                    | <composite state>
<state partition connection> ::=
                    connect <outer entry point> and <inner entry point> <end>
<outer entry point> ::=
                    <state partition identifier> via <point>
<inner entry point> ::=
                    <state partition identifier> via <point>
<point> ::=
                    <state entry point> | <state exit point> | default
```

State connection point

State connection points are defined in composite states, both directly specified composite states and state types, and represent connection points for entry and exit of a composite state.

Concrete textual grammar

```
<state connection points> ::=
                    { in <state entry points> | out <state exit points> } <end>
<state entry points> ::=
                    <state entry point>
                  | ( <state entry point> { , <state entry point> }* )
<state exit points> ::=
                    <state exit point>
                  | ( <state exit point> { , <state exit point> }* )
<state entry point> ::=
                    state entry point name>
<state exit point> ::=
                    <state exit point name>
```

Connect

Concrete textual grammar

```
<connect part> ::=
                    connect [<virtuality>] [<connect list>] <end>
                    [<on exception>] <exit transition>
<connect list> ::=
                    <state exit point name> { , <state exit point name>}*
                  | <asterisk connect list>
<exit transition> ::=
                    <transition>
<asterisk connect list> ::=

                    <asterisk>
```

Transition

Transition body

Concrete textual grammar

```
<transition> ::=
                    {<transition string> [<terminator statement>] }
                  | <terminator statement>
<transition string> ::=
                    {<action statement>}+
```

```
<action statement> ::=
                        [<label>]
                        { <action 1> <end> [ <on exception> ] | <action 2> <end> }
<action 1> ::=
                        <task>
                |       <output>
                |       <create request>
                |       <decision>
                |       <set>
                |       <reset>
                |       <export>
                |       <procedure call>
                |       <remote procedure call>
<action 2> ::=
                        <transition option>
<terminator statement> ::=
                        [<label>]
                        { <terminator 1> <end> [ <on exception> ] | <terminator 2> <end> }
<terminator 1> ::=
                        <return>
                |       <raise>
<terminator 2> ::=
                        <nextstate>
                |       <join>
                |       <stop>
```

Transition terminator

Nextstate

Concrete textual grammar

```
<nextstate> ::=
                        nextstate <nextstate body>
<nextstate body>::=
                        <state name> [<actual parameters>] | via <state entry point name> |
                |       <dash nextstate>
<dash nextstate> ::=
                        <hyphen>
                |       <history dash nextstate>
<history dash nextstate> ::=
                        <history dash sign>
```

Join

A join alters the flow in a body by expressing that the next <action statement> to be interpreted is the one which contains the same <connector name>.

Concrete textual grammar

<join> ::=
join <connector name>

Stop

Concrete textual grammar

<stop> ::=
stop

Return

Concrete textual grammar

<return> ::=
return [<expression> | **via** <state exit point>]

Raise

Concrete textual grammar

<raise> ::=
raise <raise body>
<raise body> ::=
<exception raise>
<exception raise> ::=
<exception identifier> [<actual parameters>]

Action

Task

Concrete textual grammar

<task> ::=
task <textual task body>
<textual task body> ::=
<assignment>
| nformal text>
| ompound statement>

Create

Concrete textual grammar

<create request> ::=
create <create body>

<create body> ::=
{ <agent identifier> | <agent type identifier> | **this** } [<actual parameters>]

<actual parameters> ::=
(<actual parameter list>)

<actual parameter list> ::=
[<expression>] { , [<expression>] }*

Procedure call

Concrete textual grammar

<procedure call> ::=
call <procedure call body>

<procedure call body> ::=
[**this**] { <procedure identifier> | <procedure type expression> } [<actual parameters>]

Output

Concrete textual grammar

<output> ::=
output <output body>

<output body> ::=
<signal identifier> [<actual parameters>] , <signal identifier> [<actual parameters>] }*
<communication constraints>

<destination> ::=
<pid expression> | <agent identifier> | **this**

<via path> ::=
via { <channel identifier> | <gate identifier>

Decision

Concrete textual grammar

<decision> ::=
decision <question> <end> [<on exception>]
<decision body>
enddecision

<decision body> ::=
<answer part>+ [<else part>]

<answer part> ::=
([<answer>]) <colon> [<transition>]

<answer> ::=
<range condition> | <informal text>

<else part> ::=
else <colon> [<transition>]

<question> ::=
<expression> | <informal text> | **any**

Statement list

Concrete textual grammar

<statement list> ::=
<variable definitions> <statements>

<variable definitions> ::=
{ <variable definition statement> }*

<statements> ::=
<statement>*

<statement> ::=
<empty statement>
| ompound statement>
| ssignment statement>
| lgorithm action statement>
| all statement>
| xpression statement>
| f statement>
| ecision statement>
| oop statement>
| erminating statement>
| abelled statement>
| xception statement>

<terminating statement> ::=
<return statement>
| reak statement>
| oop break statement>
| oop continue statement>
| aise statement>

Compound statement

Concrete textual grammar

<compound statement> ::=
<left curly bracket> <statement list> <right curly bracket>

Transition actions and terminators as statements

Concrete textual grammar

<assignment statement> ::=
<assignment> <end>
<algorithm action statement> ::=
<output> <end>
| reate request> <end>
| et> <end>
| eset> <end>
| xport> <end>
<return statement> ::=
<return> <end>
<raise statement> ::=
<raise> <end>
<call statement> ::=
[call] <procedure call body> <end>

Expressions as Statements

Concrete textual grammar

<expression statement> ::=
<operation application> <end>

If statement

Concrete textual grammar

<if statement> ::=
if (<Boolean expression>) <consequence statement>
[**else** <alternative statement>]
<consequence statement> ::=
<statement>
<alternative statement> ::=
<statement>

Decision statement

Concrete textual grammar

<decision statement> ::=
decision (<expression>) <left curly bracket>
<decision statement body>
<right curly bracket>
<decision statement body> ::=
<algorithm answer part>+ [<algorithm else part>]
<algorithm answer part> ::=
(<range condition>) <colon> <statement>

<algorithm else part> ::=
else <colon> <alternative statement>

Loop statement

Concrete textual grammar

<loop statement> ::=
for ([<loop clause> { ; <loop clause> }*])
<loop body statement> [**then** <finalization statement>]
<loop body statement> ::=
<statement>
<finalization statement> ::=
<statement>
<loop clause> ::=
[<loop variable indication>]
, [<Boolean expression>]
<loop step>
<loop step> ::=
[, [{ <expression> | [**call**] <procedure call body> }]]
<loop variable indication> ::=
<loop variable definition>
| variable identifier> [<is assigned sign> <expression>]
<loop variable definition> ::=
dcl <variable name> <sort> <is assigned sign> <expression>
<loop break statement> ::=
break <end>
<loop continue statement> ::=
continue <end>

Break and labelled statements

Concrete textual grammar

<break statement> ::=
break <connector name> <end>
<labelled statement> ::=
<label> <statement>

Empty Statement

Concrete textual grammar

<empty statement> ::=
<end>

Exception Statement

Concrete textual grammar

<exception statement> ::=
try <try statement> <handle statement>+
<try statement> ::=
<statement>
<handle statement> ::=
handle (<exception stimulus list>) <statement>

Timer

Concrete textual grammar

<timer definition> ::=
timer
<timer definition item> { , <timer definition item> }* **end**
<timer definition item> ::=
<timer name> [<sort list>] [<timer default initialization>]
<timer default initialization> ::=
<is assigned sign> <Duration constant expression>
<reset> ::=
reset (<reset clause> { , <reset clause> }*)
<reset clause> ::=
<timer identifier> [(<expression list>)]
<set> ::=
set <set clause> { , <set clause> }*
<set clause> ::=
([<Time expression> ,] <timer identifier> [(<expression list>)])

Exception

Concrete textual grammar

<exception definition> ::=
exception <exception definition item> { , <exception definition item> }*
<end>
<exception definition item> ::=
<exception name> [<sort list>]

Exception handler

Concrete textual grammar

<exception handler> ::=
exceptionhandler <exception handler list> <end>
[<on exception>]
<handle>*
[**endexceptionhandler** [<exception handler name>] <end>]
<exception handler list> ::=
<exception handler name> { , <exception handler name> }*

|<asterisk exception handler list>

<asterisk exception handler list> ::=

<asterisk> | (<exception handler name> { , <exception handler name> }*) |

On-Exception

Concrete textual grammar

<on exception> ::=

onexception <exception handler name> <end>

Handle

Concrete textual grammar

<handle> ::=

handle [<virtuality>] <exception stimulus list> <end>

[<on exception>] <transition>

<exception stimulus list> ::=

<exception stimulus> { , <exception stimulus> }*

| <asterisk exception stimulus list>

<exception stimulus> ::=

<exception identifier> [([<variable>] { , [<variable>] }*)]

<asterisk exception stimulus list> ::=

<asterisk>

Data definitions

Concrete textual grammar

<data definition> ::=

<data type definition>

| nterface definition>

| yntype definition>

| ynonym definition>

Data type definition

Concrete textual grammar

<data type definition> ::=

{<package use clause>}*

<type preamble> <data type heading> [<data type specialization>]

{ end> [<data type definition body> <data type closing> <end>]

| left curly bracket> <data type definition body> <right curly bracket> }

<data type definition body> ::=

{<entity in data type>}* [<data type constructor>] <operations>

[<default initialization>[<end>]]

<data type closing> ::=

{ **endvalue** | **endobject** } **type** [<data type name>]

<data type heading> ::=

```
                    { value | object } type <data type name>
                        [ <formal context parameters> ] [<virtuality constraint>]
<entity in data type> ::=
                    <data type definition>
                  | yntype definition>
                  | ynonym definition>
                  | xception definition>
<operations> ::=
                    <operation signatures>
                    <operation definitions>
<data type reference> ::=
                    <type preamble>
                    { value | object } type <data type identifier> <type reference properties>
```

Interface definition

Concrete textual grammar

```
<interface definition> ::=
                    {<package use clause>}*
                    [<virtuality>] <interface heading>
                    [<interface specialization>]
                    <end> <entity in interface>* [<interface use list>]
                    <interface closing>
                  | package use clause>}*
                  | virtuality>] <interface heading>
                    [<interface specialization>] <end>
                  | package use clause>}*
                  | virtuality>] <interface heading>
                  [ nterface specialization>] <left curly bracket>
                        <entity in interface>* [<interface use list>]
                    <right curly bracket>
<interface closing> ::=
                    endinterface [<interface name>] <end>
<interface heading> ::=
                    interface <interface name>
                        [<formal context parameters>] [<virtuality constraint>]
<entity in interface> ::=
                    <signal definition>
                  | nterface variable definition>
                  | nterface procedure definition>
                  | xception definition>
<interface use list> ::=
                    use <signal list> <end>
<interface variable definition> ::=
                    dcl <remote variable name> { , <remote variable name>}* <sort> <end>
<interface procedure definition> ::=
                    procedure <remote procedure name> <procedure signature> <end>
<interface reference> ::=
                    [<virtuality>]
                    interface <interface identifier> <type reference properties>
```

Operations

Concrete textual grammar

<operation signatures> ::=
[<operator list>] [<method list>]
<operator list> ::=
operators <operation signature> { <end> <operation signature> }* <end>
<method list> ::=
methods <operation signature> { <end> <operation signature> }* <end>
<operation signature> ::=
<operation preamble>
{ <operation name> | <name class operation> }
[<arguments>] [<result>] [<raises>]
<operation preamble> ::=
[<virtuality>] [<visibility>] | [<visibility>] [<virtuality>]
<operation name> ::=
<operation name>
| <quoted operation name>
<arguments> ::=
(<argument> { , <argument> }*)
<argument> ::=
[<argument virtuality>] <formal parameter>
<formal parameter> ::=
<parameter kind> <sort>
<argument virtuality> ::=
virtual
<result> ::=
<result sign> <sort>

Data type constructors

Concrete textual grammar

<data type constructor> ::=
<literal list>
| tructure definition>
| hoice definition>

Literals

Concrete textual grammar

<literal list> ::=
[<visibility>] **literals** <literal signature> { , <literal signature> }* <end>
<literal signature> ::=
<literal name>
| ame class literal>
| amed number>
<literal name> ::=
<literal name>
| tring name>
<named number> ::=

<literal name> <equals sign> <Natural simple expression>

Structure data types

Concrete textual grammar

<structure definition> ::=
[<visibility>] **struct** [<field list>] <end>
<field list> ::=
<field> { <end> <field> }*
<field> ::=
<fields of sort>
| ields of sort> **optional**
| ields of sort> <field default initialization>
<fields of sort> ::=
[<visibility>] <field name> { , <field name> }* <field sort>
<field default initialization> ::=
default <constant expression>
<field sort> ::=
<sort>

Choice data types

Concrete textual grammar

<choice definition> ::=
[<visibility>] **choice** [<choice list>] <end>
<choice list> ::=
<choice of sort> { <end> <choice of sort> }*
<choice of sort> ::=
[<visibility>] <field name> <field sort>

Behaviour of operations

Concrete textual grammar

<operation definitions> ::=
{ <operation definition>
| extual operation reference>
| xternal operation definition> }*
<operation definition> ::=
{<package use clause>}*
<operation heading> <end>
<entity in operation>*
<operation body>
{ **endoperator** | **endmethod** } [[<qualifier>] <operation name>] <end>
| ackage use clause>}*
<operation heading>
[<end> <entity in operation>+] <left curly bracket>
<statement list>
<right curly bracket>
<operation heading> ::=

{ **operator** | **method** } <operation preamble> [<qualifier>] <operation name>
[<formal operation parameters>]
[<operation result>] [<raises>]

<operation identifier> ::
[<qualifier>] <operation name>

<formal operation parameters> ::=
(<operation parameters> , <operation parameters> }*)

<operation parameters> ::=
[<argument virtuality>] <parameter kind> <variable name> , <variable name>}* <sort>

<entity in operation> ::=
<data definition>
| ariable definition>
| xception definition>
| elect definition>
| acro definition>

<operation body> ::=
[<on exception>] <start> { <free action> | <exception handler> }*

<operation result> ::=
<result sign> [<variable name>] <sort>

<textual operation reference> ::=
<operation heading> **referenced** <end>

<external operation definition> ::=
<operation heading> **external** <end>

Additional data definition constructs

Name class

A name class is shorthand for writing a (possibly infinite) set of literal names or operator names efined by a regular expression.

Concrete textual grammar

<name class literal> ::=
nameclass <regular expression>

<name class operation> ::=
<operation name> **in nameclass** <regular expression>

<regular expression> ::=
<partial regular expression> { [**or**] <partial regular expression> }*

<partial regular expression> ::=
<regular element> [<Natural literal name> | <plus sign> | <asterisk>]

<regular element> ::=
(<regular expression>)
| haracter string>
| regular interval>

<regular interval> ::=
<character string> <colon> <character string>

Name class mapping

A name class mapping is shorthand for defining a (possibly infinite) number of operation definitions anging over all the names in a <name class operation>. The name class mapping allows behaviour to be defined for the operators and methods defined by a <name class operation>. A name class mapping takes place when an <operation name> that occurred in a <name class operation> within an <operation signature> of the enclosing <data type definition> is used in <operation definitions> or <operation diagram>s.

Concrete textual grammar

<spelling term> ::=
spelling (<operation name>)

Restricted visibility

Concrete textual grammar

<visibility> ::=
public | **protected** | **private**

Constraint

Concrete textual grammar

<constraint> ::=
constants (<range condition>)
| <size constraint>
<range condition> ::=
<range> { , <range> }*
<range> ::=
<closed range>
| <open range>
<closed range> ::=
<constant> <colon> <constant>
<open range> ::=
<constant>
| { <equals sign>
| <not equals sign>
| <less than sign>
| <greater than sign>
| <less than or equals sign>
| <greater than or equals sign> } <constant>
<size constraint> ::=
size (<range condition>)

Synonym definition

A synonym gives a name to a constant expression that represents one of the data items of a sort.

Concrete textual grammar

```
<synonym definition> ::=
                    synonym <synonym definition item> { , <synonym definition item>
                  }*<end>
<synonym definition item> ::=
                    <internal synonym definition item>
                  | <external synonym definition item>
<internal synonym definition item> ::=
                    <synonym name> [<sort>] <equals sign> <constant expression>
<external synonym definition item> ::=
                    <synonym name> <predefined sort> external
```

Passive use of data

Expressions

Concrete textual grammar

```
<expression> ::=
                    <operand>
                  | <create expression>
                  | <value returning procedure call>
<operand> ::=
                    <operand0>
                  | <operand> <implies sign> <operand0>
<operand0> ::=
                    <operand1>
                  | <operand0> { or | xor } <operand1>
<operand1> ::=
                    <operand2>
                  | <operand1> and <operand2>
<operand2> ::=
                    <operand3>
                  | <operand2> { <greater than sign>
                    | <greater than or equals sign>
                    | <less than sign>
                    | <less than or equals sign>
                    | in } <operand3>
                  | <range check expression>
                  | <equality expression>
<operand3> ::=
                    <operand4>
                  | <operand3> { <plus sign> | <hyphen> | <concatenation sign> } <operand4>
<operand4> ::=
                    <operand5>
                  | <operand4> { <asterisk> | <solidus> | mod | rem } <operand5>
```

<operand5> ::=
[<hyphen> | **not**] <primary>

<primary> ::=
<operation application>
| <literal>
| (<expression>)
| <conditional expression>
| <spelling term>
| <extended primary>
| <active primary>
| <synonym>

<active primary> ::=
<variable access>
| <imperative expression>

<expression list> ::=
<expression> { , <expression> }*

<simple expression> ::=
<constant expression>

<constant expression> ::=
<constant expression>

Literal

Concrete textual grammar

<literal> ::=
<literal identifier>

<literal identifier> ::=
[<qualifier>] <literal name>

Synonym

Concrete textual grammar

<synonym> ::=
<synonym identifier>

Extended primary

Concrete textual grammar

<extended primary> ::=
<indexed primary>
| <field primary>
| <composite primary>

<indexed primary> ::=
<primary> (<actual parameter list>)
| <primary> <left square bracket> <actual parameter list> <right square bracket>

<field primary> ::=
<primary> <exclamation mark> <field name>
| <primary> <full stop> <field name>
| <field name>

<field name> ::=
<name>

<composite primary> ::=
[<qualifier>] <composite begin sign> <actual parameter list> <composite end sign>

Equality expression

Concrete textual grammar

<equality expression> ::=
<operand2> { <equals sign> | <not equals sign> } <operand3>

Conditional expression

Concrete textual grammar

<conditional expression> ::=
if <Boolean expression>
then <consequence expression>
else <alternative expression>
fi

<consequence expression> ::=
<expression>

<alternative expression> ::=
<expression>

Operation application

Concrete textual grammar

<operation application> ::=
<operator application>
| <method application>

<operator application> ::=
<operation identifier> [<actual parameters>]

<method application> ::=
<primary> <full stop> <operation identifier> [<actual parameters>]

Active use of data

Variable definition

Concrete textual grammar

<variable definition> ::=
dcl [exported] <variables of sort> , <variables of sort> }* <end>

<variables of sort> ::=
<variable name> [<exported as>] { , <variable name> [<exported as>] }*
<sort> [<is assigned sign> <constant expression>]

<exported as> ::=

Variable access

Concrete textual grammar

<variable access> ::=
<variable identifier>
| **this**

Assignment and assignment attempt

Concrete textual grammar

<assignment> ::=
<variable> <is assigned sign> <expression>
<variable> ::=
<variable identifier>
| <extended variable>

Extended variable

Concrete textual grammar

<extended variable> ::=
<indexed variable>
| <field variable>
<indexed variable> ::=
<variable> (<actual parameter list>)
| <variable> <left square bracket> <actual parameter list> <right square bracket>
<field variable> ::=
<variable> <exclamation mark> <field name>
| <variable> <full stop> <field name>

Default initialization

Concrete textual grammar

<default initialization> ::=
default [<virtuality>] [<constant expression>] [<end>]

Imperative expressions

Concrete textual grammar

<imperative expression> ::=
<now expression>

2.6.3 ANTLR

ANTLR, ANother Tool for Language Recognition, (formerly **PCCTS**) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing C++ or Java actions.

Why use ANTLR

- ANTLR generates recursive decent parser has good error reporting
- The parser generated by ANTLR is readable
- ANTLR is free
- ANTLR generates both Jave & C++ parser
- ANTLR can be used to generate tree parser
- ANTLR supports syntactic, semantic predicates
- Object-Oriented concepts are well supported by ANTLR
- ANTLR provides a collection of 27 option fields which could be used for designing lexer / parser
- ANTLR's syntactic predicates allow arbitrary lookahead which is a powerful way to resolve difficult parts of a syntax.

3. PROPOSED LINE OF ATTACK

Top-down approach has been adopted to attack the problem. Top-down parsing can be viewed as an attempt to construct a parse tree for the input starting from the root and creating the nodes of the parse tree in preorder.

The top-down construction of a parse tree is done by starting with the root, labeled with the starting non terminal, and repeatedly performing the following two steps:

1. At node n, labeled with non terminal A, select one of the production for A and construct children at n for the symbol on the right side of the production.
2. Find the next node at which a subtree is to be constructed.

Recursive – descent parsing is a top-down method of syntax analysis in which a set of recursive procedure are executed to process the input. One of the special form of non-backtracking recursive-descent parsing called predictive parsing (LL[k]) is used to develop the software. A predictive parser is a program consisting of a procedure for every non terminal. Each procedure does two things:

1. It decides which production to use by looking at the lookahead symbol. The production with right side α is used if the lookahead symbol is equal to α.
2. A non-terminal results in a call to the procedure for the non-terminal, and a token matching the lookahead symbol results in the next input token being read. If at some point the token in the production does not match the lookahead symbol, an error is declared.

Thus, the lookahead symbol unambiguously determines the procedure selected for each non-terminal. The sequence of procedures called in processing the input implicitly defines a parse tree for the input.

4. DETAILS OF THE PROPOSED METHODOLOGY

ANTLR 2.7.1 is used to implement SDL parser based on the grammar files, lexical rules, source code files and associated supporting files. The grammar files designed are SDLLexer.g, SDLParser.g and SDLTreeParser.g. The other files are SDLMain.cpp , InfoAST.cpp and SDLName.cpp.

SDLMain.cpp

SDLMain is the primary file used for getting the inputs from user and parse it. This is accomplished within the main() function defined.

Main Function

The main() function takes the input from command line and checks for the service required. The service could be a help facility for software usage, a version information, to print a parse tree or to compile an SDL file. Based on the service required, it performs the needed function.

When the option is to compile, the file is stored in a buffer. The lexer is instrantirated with the buffer. The parser is attached to the lexer by parsing the lexer object as argument and the method SdlSpecification is invoked to being parsing. Under no error state, the parser generates an AST .If so, the AST value is got in a RefAST object and passed to tree parser to construct a parse tree.

SDLLexer.g

SDLLexer is a grammar file containing the lexer class definition which defines the lexical rules.

SDLParser.g

SDLParser is a grammar file containing the parser class definition which defines the parser rules.

SDLTreeParser.g

SDLTreeParser is a grammar file containing the parser tree class definition for tree construction.

Lexer

A lexer (often called a scanner) breaks up an input stream of characters into vocabulary symbols for a parser, which applies a grammatical structure to that symbol stream . Lexer class SDLLexer is inherited from ANTLR Lexer class .

Lexer class Structure

Header Session ...
Class Yourlexerclass extends Lexer;
Options ...
Yourclassmembers ...
Lexer rules ...

Header Session

Header Session is used to specify the standard header files, name space and class definition that are to be included in the generated calsse file either in pre/post fashion.

The notation for Header Session is

header "<pre/post> include <file extension>"

```
{
        source code to be included
}
```

A class called SDLCharBuffer is defined to extract a stream of characters in to a token. Various methods like reportErrors(), reportMessage(), numberofErrors(), resetErrors() and versionInfo() are defined for handling exceptions. These methods are defined as members of class SDLLexer lying in the namespace SDLGrammar.

Options

Option section is used to setup the lexer, parser or common environment like specifying the language for generated files, specifying the namespace etc. For lexer the option field is used to set values for lookahead depth, literal testing, exporting vocabulary and case sensitivity as bellow :

```
options {
        k = 2;
        exportVocab = SDLLexer ;
        testLiterals = false;
        caseSensitive  = true;
}
```

Tokens

A sequence of characters having collective meaning is called tokens. In this session a list of all valid predefined SDL names are listed.

Lexer Rules

The structure of an input stream of atoms is specified by a set of mutually-referential rules. Each rule has a name, optionally a set of arguments, optionally a "throws" clause, optionally an init-action, optionally a return value, and an alternative or alternatives. Rules defined within a lexer grammar must have a name beginning with an uppercase letter. These rules implicitly match characters on the input stream instead of tokens on the token stream. Referenced grammar elements include token references (implicit lexer rule references), characters, and strings.

The basic form of an ANTLR lexer rule is:

```
Rulename [formal parameters ] returns [type id, ...]
        :       alternative_1
        |       alternative_2
        ...
        |       alternative_n
        ;
```

Refer [2.6.1] for the rule definition of SDL-2000 Lexer.

PARSER

The parser otherwise called as syntax analyzer checks for the correctness of the syntax. The over all structure of parser specification is similar to lexer except rule definition.

Header Session

Under the Header Session the exception handlers for parser class and macros for parser tree constructions are defined.

Class SDLParser

reportErrors()

The reportError function is defined to catch and report the Recognition Exception.

reportMessage()

This method is used to tell about the error along with line and column numbers at which the error has occurred.

numberofErrors()

This method keeps a count of errors.

resetErrors()

The resetErrors function initializes the error counter to zero.

Macros

The macros namely ADDVOIDCHILD, ADDCHILD and MAKEROOT are defined in order to aid in the creation of parse tree by inworking the methods add ASTChild() and makeASTRoot() of ANTLR class astFactory.

Options

Using the option clause the lookahead depth, vocabulary to export and import, error handler mode and building AST flag are set.

```
options     {
                k=3;
                importVocab = SDLLexer;
                exprotVocab  = SDLParser;
                defaultErrorHandler = true;
                buildAST = true;
            }
```

Parser Rules

Parser rules apply structure to a stream of tokens. All parser rules begins with lowercase letters. Parser rules can throw exceptions.

The basic form of an Parser rule is:

```
Rulename [formal parameters ] returns [type id, ... ] throws ex
        :       alternative_1
        |       alternative_2
        ...
        |       alternative_n
        ;
```

Refer [2.6.2] for the rule definition of SDL-2000 Parser.

Tree-parser

A tree-parser is like a parser, except that is processes a two-dimensional tree of AST nodes. Tree parsers are specified identically to parsers, except that the rule definitions may contain a special form to indicate descent into the tree. Tree grammars are collections of EBNF rules embedded with actions, semantic predicates, and syntactic predicates.

```
rule:   alternative1
        |   alternative2
    ...
        |   alternativen
        ;
```

Each alternative production has the form:

```
#( root-token child1 child2 ... childn )
```

Every rule in the grammar is defined as a method that specifies the grammatical structure to the input. When the parser needs a tokens from lexer, the method nextToken() is invoked. The tokens returned are cached in a TokenBuffer object and passed to LT() / LA() method to determine the token type and token. Once the type reference is determined, the reference is passed to corresponding match methods to recognize production element. As it walks the production, parser generates a tree based on tree grammar.

Class SDLParser

reportErrors()

The reportError function is defined to catch and report the Resignation Exception.

reportMessage()

This method is used to tell about the error along with line and column numbers at which the error has occurred.

numberofErrors()

this method keeps a count of errors.

resetErrors()

The resetErrors function initializes the error counter to zero.

Macros

The macros namely ADDVOIDCHILD, ADDCHILD and MAKEROOT are defined in order to aid in the creation of parse tree by inworking the methods add ASTChild() and makeASTRoot() of ANTLR class astFactory.

Options

Using the option clause the lookahead depth, vocabulary to export and import, error handler mode and building AST flag are set.

```
options        {
                    k=3;
```

```
        importVocab = SDLLexer;
        exprotVocab  = SDLParser;
        defaultErrorHandler = true;
        buildAST = true;
}
```

Parser Rules

Parser rules apply structure to a stream of tokens. All parser rules begins with lowercase letters. Parser rules can throw exceptions.

The basic form of an Parser rule is:

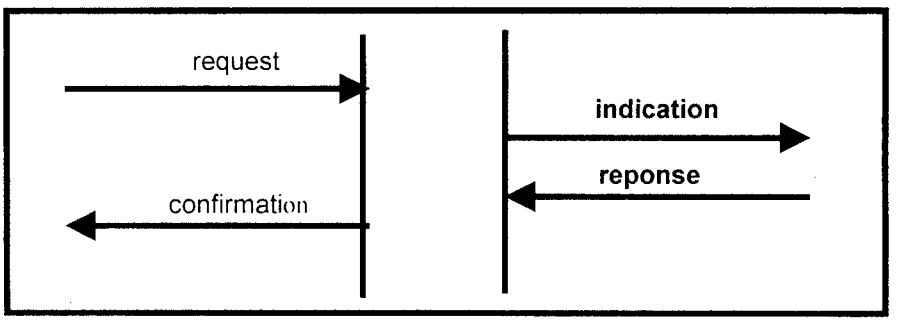
```
Rulename [formal parameters ] returns [type id, ...] throws ex
        :       alternative_1
        |       alternative_2
        ...
        |       alternative_n
        ;
```

Refer [2.6.2] for the rule definition of SDL-2000 Parser.

5. RESULTS OBTAINED

Problem Description

A system description for modeling the transport layer of V.76 protocol is taken as input. The usual conventions for signal naming in protocol is given bellow:



A request on one side is generally followed by an indication on the other side of the connection; then if the layer above accepts, it transmits a responds, translated into a confirmation on the side originator of the request. The layer operates in four different modes such as

- Exchange identification mode
- Establishment of data link connection mode
- Information transfer mode
- Release data link connection mode

In addition the layer handles encoding, segmentation, reassembly and decoding of V.76 frames

Input

The SDL / PR specification corresponding to V.76 protocol is given as input to the parser.

SDL / PR Specification

```
package V76;
signal L_EstabReq(DLCident),
  L_EstabInd(DLCident),
  L_EstabResp,
  L_EstabConf(DLCident),
  L_SetparmReq,
  L_SetparmInd,
  L_SetparmResp,
  L_SetparmConf,
  L_ReleaseReq(DLCident),
  L_ReleaseInd(DLCident),
  L_DataReq(DLCident,Idata,Integer),
  L_DataInd inherits L_DataReq;

signal
  /* V.76 commands and responses. */
  V76frame(V76paramTyp);
  /* Service User to DLC */
  signallist su2dlc=
    L_EstabReq,
    L_EstabResp,
    L_SetparmReq,
    L_SetparmResp,
    L_ReleaseReq,
    L_DataReq;

/* DLC to Service User: */
signallist dlc2su=
    L_EstabInd,
    L_EstabConf,
    L_SetparmInd,
    L_SetparmConf,
    L_ReleaseInd,
    L_DataInd;
value type V76paramTyp {
  choice
  I Iframe;
  SABME SABMEframe;
  DM DMframe;
  DISC DISCframe;
  UA UAframe;
  XIDcmd XIDframe;
  XIDresp XIDframe ;
```

```
 }

synonym maxDLC Integer = 3;
  value type SABMEframe {
    struct
      DLCi DLCident;
  }

  value type DMframe {
    struct
      DLCi DLCident;
  }

  value type DISCframe {
    struct
      DLCi DLCident;
  }

  value type UAframe {
    struct
      DLCi DLCident;
  }


  syntype XIDframe = Integer
  endsyntype;
  object type Iframe {
    struct
      DLCi DLCident;
      data Idata;
      length Integer; /* Octets in data */
      CRC Integer;
    operators
      fill_Iframe(DLCident,Idata,Integer,Integer)->Iframe;
    operator fill_Iframe(dlc DLCident,D Idata,len Integer,Crc
Integer)
   -> res Iframe;
    start;
    task res := (.dlc,D,len,Crc.);
    return res;
     endoperator;
     }
  /* maximum index for Idata. 10 for
     testing, upto 4096 normally: */

  synonym maxIdatInd Integer = 10;
  procedure CRCok(crc1 Integer)->Boolean;
    start
    comment 'Simplified Version,
    returns True if crc1
    is positive or null.';
    return crc1>=0;
  endprocedure;

  block type B76_DLC;
   gate SU
         in with (su2dlc);
   gate DL
```

```
 in with V76frame;
signal
 DLCstopped(DLCident);
channel DLCs
 from dispatch to DLC with L_DataReq,L_ReleaseReq,V76frame;
 from DLC to dispatch with DLCstopped;
     endchannel;
channel dlcSU
      from ENV via SU to dispatch with (su2dlc);
      from dispatch to ENV via SU with (dlc2su);
     endchannel;
channel dlcDL
      from dispatch to ENV via DL with V76frame;
      from ENV via DL to dispatch with V76frame;
     endchannel;
channel user
      from DLC to ENV via SU with L_DataInd, L_EstabConf;
     endchannel;
channel peer
      from DLC to ENV via DL with V76frame;
     endchannel;

process dispatch(1,1);
/* Temprary variables: */
dcl
  DLCnum, DLCpeer DLCident,
  uData Idata,
  V76para V76paramTyp,
  len Integer;
dcl
  /* To store the PIDs of the instances
     of process DLC,necessary in
     outputs to route signals: */
       DLCs DLCsArray;
     start;
  nextstate ready;
state ready;
  input V76frame (V76para);
       decision (V76para ! present) ;
     (SABME):
      task DLCp := V76para ! SABME!DLCi;
           decision DLCs(DLCp);
    (NULL):
               output L_EstabInd;
    (DLCpeer):
      nextstate waitEstabResp;
    else:
       output L_ReleaseInd(DLCpeer);
                output V76frame(DM.DLCpeer) via dlcDL;
       nextstate -;
    enddecision;
    (UA):
       output V76frame(V76para) to DLCs(V76para.UA.DLCi);
       nextstate -;
    ( I ):
       output V76frame(V76para) to DLCs(V76para.I.DLCi);
       nextstate -;
    ( DM ):
```

```
   output V76frame(V76para) to DLCs(V76para.DM.DLCi);
   nextstate -;
( DISC ):
   output V76frame(V76para) to DLCs(V76para.DISC.DLCi);
   nextstate -;
( XIDcmd ):
   output L_SetparamInd;
   nextstate waitParamResp;
( XIDresp):
   output L_SetparamConf;
   nextstate -;
else:
   nextstate -;
   enddecision;
  endstate;

       state waitEstabResp;
input L_EstabResp;
           create DLC(DLCpeer,false);
           task DLCs(DLCpeer) := offspring;
 nextstate ready;
           save V76frame;
  endstate;

       state waitParamResp;
input L_SetparamResp;
          output V76frame(XIDresp) via dlcDL;
 nextstate ready;
  endstate;

  state ready;
input L_DataReq(DLCnum,uData,len);
decision DLCs(DLCnum);
(NULL):
   /* DLCnum not setup */
else:
   output L_DataReq(DLCnum,uData,len) to DLCs(DLCnum);
enddecision;
nextstate -;

  priority input L_ReleaseReq(DLCnum);
decision DLCs(DLCnum);
(NULL):
   /* DLCnum not setup */
else:
   output L_ReleaseReq(DLCnum) to DLCs(DLCnum);
enddecision;
nextstate -;
input L_EstabReq(DLCnum);
decision DLCs(DLCnum);
(NULL):
   create DLC(DLCnum,True);
                /* 'DLCnum not used, we create an instance
                        of process DLC */
   task DLCs(DLCnum) := offspring;
                /* we store into the table PID of the
                        new instance */
else:
```

```
        output L_ReleaseInd(DLCnum);
      enddecision;
      nextstate waitUA;
        endstate;

        state ready;
      input DLCstopped(DLCnum);
        output L_ReleaseInd(DLCnum);
        task DLCs(DLCnum) := NULL;
        nextstate -;
        input L_SetparamReq;
                    output V76frame(XIdcmd) via dlcDL;
          nextstate -;
        endstate;

        state waitUA;
          input V76frame(V76para);
          decision V76para.present;
           (UA):
         output V76frame(V76para) to DLCs(V76para.UA.DLCi);
         nextstate ready;
          else:
         nextstate -;
       enddecision;
         endstate;
      endprocess;

           process DLC(0,maxDLC+1);
      timer t320 := 12.0;
      /* max.numer of retransmissions: */
      synonym N320 Integer = 3;
      dcl
        N320cnt Integer;
           dcl
             uData, Idata,len Integer, Iparam Iframe, V76para
V76paramTyp;
      start;
        decision originator;
           (True):
          task N320cnt := 0;
          retry:
                              output V76frame(SABME.me);
             set (T320);
             nextstate waitUA;
           (False):
                          output V76frame(UA.me);
          nextstate connected;
             enddecision;

        state waitUA;
          input V76frame(V76para);
          decision V76para.present;
           (UA):
          reset (T320);
          output L_Estabconf(me);
          nextstate connected;
           (DM):
```

```
      stop;
       else:
      nextstate -;
    enddecision;
  input T320;
        decision N320cnt;
             (< N320):
      task N320cnt := N320cnt + 1;
      join retry;
       else:
      output DLCstopped(me);
      stop;
    enddecision;
   endstate;

   state connected;
     input L_DataReq(,uData,len);
     task Iparam := fill_Iframe(me,uData,len,15);
          output V76frame(I.Iparam) via peer;
     nextstate -;
     input L_ReleaseReq;
            output V76frame(DISC.me) via peer;
       nextstate waitUAdisc;
     input V76frame(V76para);
       decision V76para.present;
   ( DISC ):
                      output V76frame(UA.me) via peer;
      output DLCstopped(me);
      stop;
   ( I ):
      decision call CRCok(V76para.I.CRC);
      ( True ):
         output L_DataInd(me,V76para.I.data,
            V76para.I.length);
      ( False ):
      enddecision;
      nextstate -;
   else:
      nextstate -;
   enddecision;
       endstate;

       state waitUAdisc;
     input V76frame(V76para);
     decision V76para.present;
     ( UA,DM ):
       output DLCstopped(me);
       stop;
     else:
       nextstate -;
    enddecision;
       endstate;
      endprocess;
    endblock type;
  endpackage;

use V76;
package framing;
```

```
signal
   L2DataReq(L2Data),L2DataInd(L2Data);
synonym maxL2datInd Integer = 3;

     block type framing;
  gate DLC
   in with V76frame;
  gate L2
   in with L2DataInd;
  synonym maxencodV76ind Integer = maxIdatInd*4;
       object type conversions {
         operators
           Octet2Int(Octet)->Integer;
           Int2Octet(Integer)->Octet;
       }
  channel segL2 from segmented to ENV via L2 with L2DataReq;
       endchannel;
  channel reasL2 from ENV via L2 to reassember with L2DataInd;
       endchannel;
  channel segDLC from ENV via DLC to segmenter with V76frame;
       endchannel;
  channel reasDLC from reassembler to ENV via DLC with V76frame;
       endchannel;

  process segmenter;
    dcl
       V76para V76paramTyp,
       eV76 encodedV76,
       octNr Integer,
       trailOctetNr Integer,
       fullCellsNr Integer,
       n Integer,
       isLastCell Integer;
    procedure encodePDU(inPDU V76paramTyp)->outBuffer encodedV76;
      dcl
    j Integer;
      start;
    task outBuffer := (.'00'h.);
    decision inPDU.present;
      ( XIDcmd ):
       task outBuffer(0) := '01'h;
      ( XIDresp ):
       task outBuffer(0) := '02'h;
      ( SABME ):
       task outBuffer(0) := '03'h;
       task outBuffer(1) := int2Octet(inPDU.SABME.DLCi);
      ( UA ):
       task outBuffer(0) := '04'h;
       task outBuffer(1) := int2Octet(inPDU.UA.DLCi);
      ( DM ):
       task outBuffer(0) := '05'h;
       task outBuffer(1) := int2Octet(inPDU.DM.DLCi);
      ( DISC ):
       task outBuffer(0) := '06'h;
       task outBuffer(1) := int2Octet(inPDU.DISC.DLCi);
      ( I ):
       task outBuffer(0) := '07'h;
```

```
                        task outBuffer(2) := int2Octet(inPDU.I.CRC);
                        task outBuffer(3) :=
int2Octet(inPDU.I.length);
        task j := 0;
       next1:
        decision j < inPDU.I.length;
          ( True ):
           task outBuffer(j+4) := inPDU.I.data(j);
                                  task  j := j+1;
           join next1;
        enddecision;
       enddecision;
       return;
      endprocedure;

      procedure sendCell(eV encodedV76, frm Integer,
          nb Integer, lastCell Integer);
        dcl
          k Integer,
          cell L2data;
        start;
           task cell := ('00'h.);
           task cell(0) := int2Octet(lastCell);
           task cell(1) := if lastCell=1
           then int2Octet(octNr) else '00'h fi;
           task k := 0;
           next1:
        decision k<nb;
           ( True ):
           task cell(k+2) := eV(frm+k);
           task k := k+1;
                              join next1;
           ( False ):
           output L2DataReq(cell);
           return;
           enddecision;
        endprocedure;

        start;
          nextstate ready;
        state ready;
          input V76frame(V76para);
          task eV76 := call encodePDU(V76para);
          task octNr := if V76para.present = 1
                        then 4+V76para.I.length
                        else 2 fi;
                        task trailOctetNr := octNr
mod(maxL2datInd+1-2);
          task fullCellsNr := octNr/(maxL2DatInd+1-2);
          task n := 0;
          next1:
            decision n < fullCellsNr;
              ( True ):
             task isLastCell := if (n = fullCellsNr-1)
             and trailOctetNr = 0 then 1 else 0 fi;
             call sendCell(eV76,n*(maxL2datInd-1),
             maxL2datInd-1,isLastCell);
             task n := n + 1;
```

```
            join next1;
             ( False ):
            decision trailOctetNr;
              ( 0 ):
              else:
             call sendCell(eV76,n'(maxLXGarInd-1),
             trailOctetNr,1);
            enddecision;
            nextstate -;
             enddecision;
           endstate;
      endprocess;

    process reassembler;
      dcl
        m Integer,
        cell L2data,
        len Integer,
        cellNr Integer,
        eV76 encodedV76;
      procedure decodePDU(inBuffer encodedV76)-
          outPDU V76paramTyp;
        dcl
      Istruct Iframe,
      n Integer;
        start;
      decision inBuffer(0);
        ( '01'h ):
                        task outPDU := XIDcmd.0;
        ( '02'h ):
                        task outPDU := XIDresp.0;
        ( '03'h ):
                        task outPDU := SABME.Octet2Int(inBuffer(1));
        ( '04'h ):
                        task outPDU := UA.Octet2Int(inBuffer(1));
        ( '05'h ):
                        task outPDU := DM.Octet2Int(inBuffer(1));
        ( '06'h ):
                        task outPDU := DISC.Octet2Int(inBuffer(1));
        ( '07'h ):
                        task Istruct.DLCi
:=(.Octet2Int(inBuffer(1)).);
                        task Istruct.CRC :
(.Octet2Int(inBuffer(2)).);
                        task Istruct.length
:=(.Octet2Int(inBuffer(3)).);
         task n := 0;
         next1:
           decision n<Istruct.length;
             ( True ):
            task Istruct.data(n) :=
            inBuffer(n+4);
            task n := n + 1;
            join next1;
             ( False ):
           enddecision;
                        task outPDU := 1.Istuct;
         enddecision;
```

```
  return;
endprocedure;

start;
            task cellNr := 0;
            task eV7 :=  (. '00'h .);
 nextstate ready;
state ready;
   input L2DataInd(cell);
   task m := 2;
   next1:
     decision m < maxL2datInd+1;
  ( True ):
     task eV76(m-2+cellNr*(maxL2datInd-1))
     := cell(m);
     task m := m + 1;
     join next1;
  ( False ):
                          task cellNr := cellNr + 1;
     decision cell(0) = '01'h;
       ( False ):
       ( True ):
        task len := octet2Int(cell(1));
        output V76frame(call decodePDU
                 (eV76));
                                  task cellNr := 0;
                                  task eV76 := (. '00'h .);
     enddecision;
     nextstate -;
  enddecision;
     endstate;
             endprocess ;
             endblock type framing;
           endpackage;

   use V76; use framing;
   system stackTest;
   channel DLCaSU
from ENV to DLCa via SU with (su2dlc);
from DLCa via SU to ENV with (dlc2su);
   endchannel;
   channel DLCaDL
from DLCa via DL to framingA via DLC with V76frame;
from framingA via DLC to DLCa via DL with V76frame;
   endchannel;
   channel DLCbDL
from DLCb via DL to framingB via DLC with V76frame;
from framingB via DLC to DLCb via DL with V76frame;
   endchannel;
   channel DLCbSU
from ENV to DLCb via SU with (su2dlc);
from DLCb via SU to ENV with (dlc2su);
   endchannel;
   channel FEDphyA
from framingA via L2 to layer2stub with L2DataReq;
from layer2stub to framingA via L2 with L2DataInd;
   endchannel;
   channel FEDphyB
```

```
      from framingB via L2 to layer2stub with L2DataReq;
      from layer2stub to framingB via L2 with L2DataInd;
         endchannel;
         block DLCa : V76_DLC;
         block DLCb : V76_DLC;
         block framingA : framing;
         block framingB : framing;
         block layer2stub;
         channel fromA
      from ENV to AtoB via gIn  with L2DataReq;
              endchannel;
         channel fromB
      from ENV to BtoA via gIn  with L2DataReq;
              endchannel;
         channel toA
      from BtoA via gOut  to ENV with L2DataInd;
              endchannel;
         channel toB
      from AtoB via gOut  to ENV with L2DataInd;
              endchannel;
         connect FEDphyA and fromA, toA;
         connect FEDphyB and fromB, toB;
         process type toPeer;
      gate gIn
         in with L2DataReq;
      gate gOut
         out with L2DataInd;
      dcl
         L2par L2Data;
      start;
        nextstate ready;
      state ready;
        input L2DataReq(L2par);
        decision 'Lose the frame ?';
          ( 'No' ):
         output L2DataInd(L2par) via gOut;
        enddecision;
        nextstate -;
      endstate;
         endprocess type;
         process AtoB : toPeer;
         process BtoA : toPeer;
       endblock;
     endsystem;
/* end */
```

Output

The system checks for the correctness of the syntactical structure of the given SDL / PR specification. Any deviation in the grammatical structure found is reported as an error with appropriate diagnostic message. Since the given input is bug free the parser reports with zero errors. To show the working of parser some of the sample output showing the errors displayed by the parser is given.

Sample Outputs

Test case 1 :

The following error is reported by the parser when the end statement is not given for previous line number.

```
SDL-2000 Parser, compiled on Oct 28 2001

Running the SDL-2000 parser...

dll.pr ; 15;unexpected token; signal

1 error(s) found.
```

Test case 2 :

The following errors are reported by the parser when the end statement is not given for line number 35 and the keyword nextstate is misspelled at line number 160.

```
SDL-2000 Parser, compiled on Oct 28 2001

Running the SDL-2000 parser...

Testdll.pr 36; unexpected token; choice
Testdll.pr 38; unexpected token; SABME
Testdll.pr 44; unexpected token;}
Testdll.pr 46; unexpected token;=
Testdll.pr 46; unexpected token;3
Testdll.pr 160; expecting ":", found '-'
Testdll.pr 160; unexpected token: ;

7 error(s) found.
```

6. CONCLUSIONS AND FUTURE OUTLOOK

In this project a parser has been successfully built for the specification language SDL – 2000. The parser covers all the features of SDL 2000 such as structural concepts, system behaviour, agents, data concept and generic system definition. The parser developed is capable of parsing any SDL specification given in SDL –2000 phrase notation and the specifications can be of any length. When the option to display AST is specified , it displays the abstract syntax tree In other cases it produces an error report specifying the line and column numbers at which the errors have occurred along with the details of the errors encountered during parsing. If no errors, the parser ends successfully with no errors. In future the project could be extended to a SDL - 2000 compiler by adding a code generator.

7. REFERENCES

Books

- Alfred V Aho, Ravi Sethi, Jaffrey D. Ullman . "**Compilers Principles , Techniques and Tools** ", Wesley publishing company , 1986
- Laurent Doldi . "**SDL Illustrated Visually Design Executable Models**", British Library Cataloguing in Publication Data , 2001.
- Terence Parr , "**ANTLR Reference Manual**"
- ITU-T Recommendation Z.100

Web Sites

- **perso.wanadoo.fr/doldi/sdl**
- **www.antlr.org**
- **www.sdl-forum.org**

8. APPENDICES

8.1 Abbreviations used in SDL / PR specifications

DISC	DISConnect
DLC	Data Link Connection entry
DM	Disconnect Mode
I	Information
SABME	Set Asynchronous Balanced Mode Extended
SU	Service User
XID	eXchange IDentification
UA	Unnumbered Acknowledge

8.2 The following table summarizes punctuation and keywords in ANTLR.

Symbol	**Description**
(...)	subrule
(...)*	closure subrule
(...)+	positive closure subrule
(...)?	optional
{...}	semantic action
[...]	rule arguments
{...}?	semantic predicate
(...)=>	syntactic predicate
|	alternative operator
..	range operator
~	not operator
.	wildcard
=	assignment operator
:	label operator, rule start
;	rule end
<...>	element option
class	grammar class
extends	specifies grammar base class

`returns`	specifies return type of rule
`options`	options section
`tokens`	tokens section
`header`	header section
`tokens`	token definition section

SDLMain.cpp

```
#include <vector>
#include <fstream>
#include "antlr/AST.hpp"
#include "antlr/ANTLRException.hpp"
#include "SDLLexer.hpp"
#include "SDLParser.hpp"
#include "SDLTreeParser.hpp"

using std::istream;
using std::ifstream;
using std::cout;
using std::cin;
using std::cerr;
using std::endl;
using std::vector;
using std::string;
using antlr::RefAST;
using antlr::InfoAST;


void printAST( RefAST node, const vector<string> &tokenNames,
               bool flatten, int level = 0 )
{
  RefAST          node2;
  const InfoAST   *infoAST;

  for ( node2 = node; node2 != 0; node2 = node2 -> getNextSibling() )
  {
    if ( node2 -> getFirstChild() != antlr::nullAST )
    {
      flatten = false;
      break;
    }
  }

  for ( node2 = node; node2 != 0; node2 = node2 -> getNextSibling() )
  {
    if ( !flatten || node2 == node )
    {
      for ( int i = 0; i < level; i++ )
        cout << "  ";
    }

    if ( node2 -> getText() != "" )
      cout << "'" << node2 -> getText() << "' ("
           << tokenNames[ node2 -> getType() ] << ")";
    else
      cout << tokenNames[ node2 -> getType() ];

    if ( ( infoAST = dynamic_cast< const InfoAST * >( node2.get() ) ) != 0 )
    {
      if ( infoAST -> getLine() != 0 )
        cout << " (" << infoAST -> getLine() << "," << infoAST -> getColumn() << ")";
    }

    if ( flatten )
      cout << " ";
    else
      cout << endl;

    if ( node2 -> getFirstChild() != antlr::nullAST )
      printAST( node2 -> getFirstChild(), tokenNames, flatten, level + 1 );
  }

  if ( flatten )
    cout << endl;
}
```

```
struct
{
  bool            isHelp;
  bool            isVersion;
  bool            isReadStdin;
  bool            isPrintTree;
  bool            isNoFlattening;
  vector< int >   fileNameArgs;
} options = { false, false, false, false, false, vector< int >() };


int main( int argc, char *argv[] )
{
  for ( int i = 1; i < argc; i++ )
  {
    if ( strcmp( "--help", argv[ i ] ) == 0 || strcmp( "-h", argv[ i ] ) == 0 )
    {
      options.isHelp = true;
      continue;
    }

    if ( strcmp( "--version", argv[ i ] ) == 0 || strcmp( "-v", argv[ i ] ) == 0 )
    {
      options.isVersion = true;
      continue;
    }

    if ( strcmp( "--read-stdin", argv[ i ] ) == 0 || strcmp( "-s", argv[ i ] ) == 0 )
    {
      options.isReadStdin = true;
      continue;
    }

    if ( strcmp( "--print-tree", argv[ i ] ) == 0 || strcmp( "-t", argv[ i ] ) == 0 )
    {
      options.isPrintTree = true;
      continue;
    }

    if ( strcmp( "--no-flattening", argv[ i ] ) == 0 || strcmp( "-f", argv[ i ] ) == 0 )
    {
      options.isNoFlattening = true;
      continue;
    }

    if ( strncmp( "--", argv[ i ], 2 ) == 0 )
    {
      cerr << endl
           << "Error: Invalid command line option '" << argv[ i ] << "'." << endl
           << endl;
      exit( 1 );
    }

    options.fileNameArgs.push_back( i );
  }

  cout << "SDL-2000 Parser, compiled on " << __DATE__ << endl
       << endl;
  if ( options.isHelp )
  {
    cout << "Usage:" << endl
         << endl
         << "  SDL2000Parser [ options ] [ file ... ]" << endl
         << endl
         << "Options:" << endl
         << endl
         << "  -h, --help                display this overview and exit" << endl
         << "  -v, --version             display program version and exit" << endl
         << "  -s, --read-stdin          read from standard input" << endl
         << "  -t, --print-tree          print abstract syntax tree" << endl
         << "  -f, --no-flattening       disable flattened output of AST" << endl
         << endl;
```

```
  }

  if ( options.isVersion )
  {
    exit( 0 );
  }

  if ( options.fileNameArgs.empty() && !options.isReadStdin )
  {
    cout << "Usage:" << endl
         << endl
         << "  SDL2000Parser [ --help ] [ --version ] [ --read-stdin ] [ --print-tree ]"
<< endl
         << "                [ --no-flattening ] [ file ... ]" << endl
         << endl;
    exit( 0 );
  }

  if ( options.isReadStdin && options.fileNameArgs.size() > 0 )
  {
    cerr << "Error: Cannot read from 'stdin' and from files at the same time." << endl
         << endl;
    exit( 1 );
  }

  for ( unsigned int i = 0; i < options.fileNameArgs.size() || ( options.isReadStdin && i
== 0 ); i++ )
  {
    ifstream   *file = 0;

    if ( !options.isReadStdin )
    {
      file = new ifstream( argv[ options.fileNameArgs[ i ] ] );

      if ( !*file )
      {
        cerr << "Error: Cannot open file '" << argv[ options.fileNameArgs[ i ] ] << "'."
<< endl
             << endl;
        exit( 1 );
      }
    }

    try
    {
      unsigned int                noOfErrors;
      SDLGrammar::SDLCharBuffer   buffer( ( file != 0 ) ? *dynamic_cast< istream * >( file
) : cin );
      SDLGrammar::SDLLexer        lexer( buffer );
      SDLGrammar::SDLParser       parser( lexer );
      SDLGrammar::SDLTreeParser   treeparser;

      lexer.setFilename( ( file != 0 ) ? argv[ options.fileNameArgs[ i ] ] : "<stdin>" );
      parser.setFilename( ( file != 0 ) ? argv[ options.fileNameArgs[ i ] ] : "<stdin>" );

      lexer.resetErrors();
      parser.resetErrors();

      parser.setASTNodeFactory( &antlr::InfoAST::factory );

      if ( options.fileNameArgs.size() > 1 )
        cout << "*** Input file: " << argv[ options.fileNameArgs[ i ] ] << " ***" << endl
             << endl;

      cout << "Running the SDL-2000 parser..." << endl
           << endl;

      parser.sdlSpecification();
```

```
    if ( noOfErrors == 0 && tree != antlr::nullAST )
    {
      cout << "Running the SDL-2000 tree parser (no output)... " << endl
           << endl;

      treeparser.sdlSpecification( tree );

      cout << "... finished." << endl
           << endl;
    }

    // noOfErrors = noOfErrors + treeparser.numberOfErrors()

    cout << noOfErrors << " error(s) found." << endl << endl;

    if ( noOfErrors > 0 )
      continue;

    if( options.isPrintTree )
    {
      if ( tree != antlr::nullAST )
      {
        cout << "Printing the abstract syntax tree..." << endl
             << endl;

        printAST( tree, parser.getTokenNames(), !options.isNoFlattening );

        cout << endl;
      }
      else
      {
        cout << "No tree generated." << endl
             << endl;
      }
    }
  }
  catch( antlr::ANTLRException& e )
  {
    cerr << e.toString() << endl;
  }

  if ( file != 0 )
    delete file;
 }
}
```

SDLLexer.g

```
header "pre_include_hpp"
{
  // $Id: SDLLexer.g,v 1.11 2001/06/02 23:21:18 schmitt Exp $

  #include <string>
  #include "antlr/CharBuffer.hpp"

  using std::string;
}


header "post_include_hpp"
{
  namespace SDLGrammar
  {
    class SDLCharBuffer : public antlr::CharBuffer
    {
      public:

        SDLCharBuffer( std::istream &input) : CharBuffer( input )
        {
        }

        int getChar()
        {
                int   c;
          static int   nextc = 0;

          if ( nextc != 0 )
          {
            c = nextc;
            nextc = 0;
          }
          else
            c = CharBuffer::getChar();

          if ( c != '_' )
            return c;

          nextc = CharBuffer::getChar();
          if ( nextc < 0 || nextc > 32 )
            return c;

          do
          {
            nextc = CharBuffer::getChar();
          } while( nextc >= 0 && nextc <= 32 );

          return getChar();
        }
    };
  }
}


header "post_include_cpp"
{
  #include <iostream>

  using std::cerr;
  using std::endl;
```

```
namespace SDLGrammar
{
  void SDLLexer::reportError( const string &errorMessage )
  {
    reportMessage( errorMessage );
    noOfErrors++;
  }

  void SDLLexer::reportMessage( const string &message ) const
  {
    if ( getFilename().length() > 0 )
      cerr << getFilename();
    else
      cerr << "<stdin>";

    cerr << ":" << getLine() << ":" << getColumn() << ": error: " << message << endl;
  }

  unsigned int SDLLexer::numberOfErrors() const
  {
    return noOfErrors;
  }

  void SDLLexer::resetErrors()
  {
    noOfErrors = 0;
  }

  const string &SDLLexer::versionInfo()
  {
    static string   info = "$Id: SDLLexer.g,v 1.11 2001/06/02 23:21:18 schmitt Exp $";

    return info;
  }
 }
}


options
{
  language  = "Cpp";
  namespace = "SDLGrammar";
  namespaceStd = "std";
  namespaceAntlr = "antlr";
  genHashLines = true;
}


class SDLLexer extends Lexer;

  options
  {
    k = 2;
    exportVocab = SDLLexer;
    testLiterals = false;
    caseSensitive = true;
  }

  tokens
  {
    ABSTRACT            = "abstract";
    ACTIVE              = "active";
    ADDING              = "adding";
    AGGREGATION         = "aggregation";
    ALTERNATIVE         = "alternative";
    AND                 = "and";
    ANY                 = "any";
```

```
AS                      = "as";
ASSOCIATION             = "association";
ATLEAST                 = "atleast";
BLOCK                   = "block";
BREAK                   = "break";
CALL                    = "call";
CHANNEL                 = "channel";
CHOICE                  = "choice";
COMMENT                 = "comment";
COMPOSITION             = "composition";
CONNECT                 = "connect";
CONNECTION              = "connection";
CONSTANTS               = "constants";
CONTINUE                = "continue";
CREATE                  = "create";
DCL                     = "dcl";
DECISION                = "decision";
DEFAULT                 = "default";
ELSE                    = "else";
ENDALTERNATIVE          = "endalternative";
ENDBLOCK                = "endblock";
ENDCHANNEL              = "endchannel";
ENDCONNECTION           = "endconnection";
ENDDECISION             = "enddecision";
ENDEXCEPTIONHANDLER     = "endexceptionhandler";
ENDINTERFACE            = "endinterface";
ENDMACRO                = "endmacro";
ENDMETHOD               = "endmethod";
ENDOBJECT               = "endobject";
ENDOPERATOR             = "endoperator";
ENDPACKAGE              = "endpackage";
ENDPROCEDURE            = "endprocedure";
ENDPROCESS              = "endprocess";
ENDSELECT               = "endselect";
ENDSTATE                = "endstate";
ENDSUBSTRUCTURE         = "endsubstructure";
ENDSYNTYPE              = "endsyntype";
ENDSYSTEM               = "endsystem";
ENDTYPE                 = "endtype";
ENDVALUE                = "endvalue";
ENV                     = "env";
EXCEPTION               = "exception";
EXCEPTIONHANDLER        = "exceptionhandler";
EXPORT                  = "export";
EXPORTED                = "exported";
EXTERNAL                = "external";
FI                      = "fi";
FINALIZED               = "finalized";
FOR                     = "for";
FROM                    = "from";
GATE                    = "gate";
HANDLE                  = "handle";
IF                      = "if";
IMPORT                  = "import";
IN                      = "in";
INHERITS                = "inherits";
INPUT                   = "input";
INTERFACE               = "interface";
JOIN                    = "join";
LITERALS                = "literals";
MACRO                   = "macro";
MACRODEFINITION         = "macrodefinition";
MACROID                 = "macroid";
METHOD                  = "method";
METHODS                 = "methods";
MOD                     = "mod";
NAMECLASS               = "nameclass";
```

```
NEXTSTATE           = "nextstate";
NODELAY             = "nodelay";
NONE                = "none";
NOT                 = "not";
NOW                 = "now";
OBJECT              = "object";
OFFSPRING           = "offspring";
ONEXCEPTION         = "onexception";
OPERATOR            = "operator";
OPERATORS           = "operators";
OPTIONAL            = "optional";
OR                  = "or";
ORDERED             = "ordered";
OUT                 = "out";
OUTPUT              = "output";
PACKAGE             = "package";
PARENT              = "parent";
PRIORITY            = "priority";
PRIVATE             = "private";
PROCEDURE           = "procedure";
PROTECTED           = "protected";
PROCESS             = "process";
PROVIDED            = "provided";
PUBLIC              = "public";
RAISE               = "raise";
REDEFINED           = "redefined";
REFERENCED          = "referenced";
REM                 = "rem";
REMOTE              = "remote";
RESET               = "reset";
RETURN              = "return";
SAVE                = "save";
SELECT              = "select";
SELF                = "self";
SENDER              = "sender";
SET                 = "set";
SIGNAL              = "signal";
SIGNALLIST          = "signallist";
SIGNALSET           = "signalset";
SIZE                = "size";
SPELLING            = "spelling";
START               = "start";
STATE               = "state";
STOP                = "stop";
STRUCT              = "struct";
SUBSTRUCTURE        = "substructure";
SYNONYM             = "synonym";
SYNTYPE             = "syntype";
SYSTEM              = "system";
TASK                = "task";
THEN                = "then";
THIS                = "this";
TIMER               = "timer";
TO                  = "to";
TRY                 = "try";
TYPE                = "type";
USE                 = "use";
VALUE               = "value";
VIA                 = "via";
VIRTUAL             = "virtual";
WITH                = "with";
XOR                 = "xor";

ResultSign                  = "->";
CompositeBeginSign          = "(.";
CompositeEndSign            = ".)";
ConcatenationSign           = "//";
```

```
    HistoryDashSign             = "-*";
    GreaterThanOrEqualsSign     = ">=";
    ImpliesSign                 = "=>";
    IsAssignedSign              = ":=";
    LessThanOrEqualsSign        = "<=";
    NotEqualsSign               = "/=";
    QualifierBeginSign          = "<<";
    QualifierEndSign            = ">>";

    ExclamationMark             = "!";
    QuotationMark               = "\"";
    LeftParenthesis             = "(";
    RightParenthesis            = ")";
    Asterisk                    = "*";
    PlusSign                    = "+";
    Comma                       = ",";
    Hyphen                      = "-";
    FullStop                    = ".";
    Solidus                     = "/";
    Colon                       = ":";
    Semicolon                   = ";";
    LessThanSign                = "<";
    EqualsSign                  = "=";
    GreaterThanSign             = ">";
    LeftSquareBracket           = "[";
    RightSquareBracket          = "]";
    LeftCurlyBracket            = "{";
    RightCurlyBracket           = "}";
    NumberSign                  = "#";
    DollarSign                  = "$";
    PercentSign                 = "%";
    Ampersand                   = "&";
    Apostrophe                  = "'";
    QuestionMark                = "?";
    CommercialAt                = "@";
    ReverseSolidus              = "\\";
    CircumflexAccent            = "^";
    Underline                   = "_";
    GraveAccent                 = "`";
    VerticalLine                = "|";
    Tilde                       = "~";
  }

{
  public:

    unsigned int numberOfErrors() const;
    void resetErrors();

    static const string &versionInfo();

  protected:

    void reportError( const string &errorMessage );
    void reportMessage( const string &message ) const;

  private:

    unsigned int   noOfErrors;
}

LexicalUnit :

    Name                        { $setType(testLiteralsTable(Name)); }
  | QuotedOperationName         { $setType(QuotedOperationName); }
  | ( BitString ) =>
    BitString                   { $setType(BitString); }        // Test 'BitString' ...
```

```
  | ( HexString ) =>
    HexString                   { $setType(HexString) }             and 'HexString' first
  | CharacterString             { $setType(CharacterString) }
  | ( "/*" ) =>
    Note                        { $setType(antlr::Token::SKIP); }
  | ( CompositeSpecial ) =>     // no warning despite ambiguity due to bug in ANTLR!
    CompositeSpecial            { $setType(testLiteralsTable(CompositeSpecial)); }
  | Special                     { $setType(testLiteralsTable(Special)); }

  ;

protected Name :

    ( ( DecimalDigit )+ '.' ) =>
    ( DecimalDigit )+ ( '.' ( DecimalDigit )+ )*
  | ( '_' )* Word ( '_' ( Alphanumeric )* )*

  ;

protected Word :

  ( Alphanumeric )+

  ;

protected Alphanumeric :

    Letter
  | DecimalDigit

  ;

protected Letter :

    UppercaseLetter
  | LowercaseLetter

  ;

protected UppercaseLetter :

  ( 'A' .. 'Z' )

  ;

protected LowercaseLetter :

  ( 'a' .. 'z' )

  ;

protected DecimalDigit :

  ( '0' .. '9' )

  ;

protected QuotedOperationName :

  // Problem: "-" is in both InfixOperationName and MonadicOperationName

  '"'!
  (   ( InfixOperationName ) =>
      InfixOperationName
    | MonadicOperationName )
  '"'!
```

```
;

protected InfixOperationName :

    "or"   | "OR"
  | "xor"  | "XOR"
  | "and"  | "AND"
  | "in"   | "IN"
  | "mod"  | "MOD"
  | "rem"  | "REM"
  | '+'    | '-'   | '*'
  | "/="   | "<="  | ">="
  | "//"   | "=>"  | '/'
  | '='    | '>'   | '<'

  ;

protected MonadicOperationName :

    '-'
  | "not"
  | "NOT"

  ;

protected CharacterString :

  '\''!
  (   GeneralTextCharacter
    | Special
    | ( '\'' '\'' ) =>
      '\'' '\''! )*
  '\''!

  ;

protected HexString :

  '\''!
  (   DecimalDigit
    | 'a' .. 'f'
    | 'A' .. 'F' )*
  '\''! ( 'H'! | 'h'! )

  ;

protected BitString :

  '\''!
  (   '0'
    | '1' )*
  '\''! ( 'B'! | 'b'! )

  ;

protected Note :

  '/'! '*'! NoteText '*'! '/'!

  ;

protected NoteText :

  ( options { greedy = false; } :
    (   GeneralTextCharacter
      | OtherSpecial
      | '*'
```

```
      | '/'
      | '\'' ) )*

  ;

protected GeneralTextCharacter :

    Alphanumeric
  | OtherCharacter
  | Space

  ;

protected CompositeSpecial :

    "->" | "(."
  | ".)" | "//"
  | "-*" | ">="
  | "=>" | ":="
  | "<=" | "/="
  | "<<" | ">>"

  ;

protected Special :

    '/'
  | '*'
  | OtherSpecial

  ;

protected OtherSpecial :

    '!' | '#' | '(' | ')' | '+' | ',' | '-'
  | '.' | ':' | ';' | '<' | '=' | '>'
  | '[' | ']' | '{' | '}'

  ;

protected OtherCharacter :

    '"'  | '$' | '%' | '&' | '?' | '@'
  | '\\' | '^' | '_' | '`' | '|' | '~'

  ;

protected Space :

    '\000' .. '\011'      // Control characters
  | '\n' { newline(); }
  | '\013' .. '\037'      // Control characters
  | ' '

  ;

WhiteSpace :

  ( Space )+

  { $setType(antlr::Token::SKIP); }

  ;
```

SDLParser.g

```
header "pre_include_hpp"
{
  #include <string>
  #include "InfoAST.hpp"
  #include "SDLName.h"

  using std::string;
}


header "post_include_cpp"
{
  #include <iostream>

  using std::cerr;
  using std::endl;

  namespace SDLGrammar
  {
    void SDLParser::reportError( const string &errorMessage )
    {
      reportMessage( errorMessage );
      noOfErrors++;
    }

    void SDLParser::reportError( const antlr::RecognitionException &ex )
    {
      cerr << ex.toString().c_str() << endl;
      noOfErrors++;
    }

    void SDLParser::reportMessage( const string &message )
    {
      if ( getFilename().length() > 0 )
        cerr << getFilename();
      else
        cerr << "<stdin>";

      cerr << ":" << LT( 1 ) -> getLine() << ":" << LT( 1 ) -> getColumn() << ": error: " <<
message << endl;
    }

    unsigned int SDLParser::numberOfErrors() const
    {
      return noOfErrors;
    }

    void SDLParser::resetErrors()
    {
      noOfErrors = 0;
    }

    const string &SDLParser::versionInfo()
    {
      static string   info = "SDLParser.g v 1.00";

      return info;
    }
  }

  #define ADDVOIDCHILD(tokenast)      astFactory.addASTChild( currentAST, tokenast )
  #define ADDCHILD(tokenAST,token) { const antlr::RefAST &tmpAST = tokenAST; \
                                     SETLINECOLUMN( tmpAST, token ); \
                                     astFactory.addASTChild( currentAST, tmpAST ); }
```

```
  #define MAKEROOT(tokenAST,token) { const antlr::RefAST &tmpAST = tokenAST; \
                                     SETLINECOLUMN( tmpAST, token ) \
                                     astFactory.makeASTRoot( currentAST, tmpAST ); }
  #define SETLINECOLUMN(AST,token) { antlr::RefInfoAST( AST ) -> setLine( token -> getLine()
); \
                                     antlr::RefInfoAST( AST ) -> setColumn( token ->
getColumn() ); }
}


options
{
  language  = "Cpp";
  namespace = "SDLGrammar";
  namespaceStd = "std";
  namespaceAntlr = "antlr";
  genHashLines = false;         // too many compiler warnings when generating #line
statements
}


class SDLParser extends Parser;

options
{
  k = 3;
  importVocab = SDLLexer;
  exportVocab = SDLParser;
  defaultErrorHandler = true;
  buildAST = true;
  codeGenMakeSwitchThreshold = 10;
}

  // The following section defines tokens to be used for new root nodes in the AST
  //
  // Tokens that are not allowed in general for ambiguity reasons:
  //
  //   ActivePrimary, AttributeProperty, BaseType, BasicSort, BasicState, BehaviourProperty,
  //   CallStatement, CompositeStateApplication, CompositeStateList, ExceptionProperty,
Expression
  //   ExpressionStatement, FieldName, FieldProperty, FormalParameter, InformalText,
  //   InterfaceVariableProperty, Literal, LiteralIdentifier, LiteralName, Operand, Operand0,
  //   Operand1, Operand2, Operand3, Operand4, Operand5, OperationApplication,
OperationIdentifier,
  //   OperationName, OperationProperty, OperatorApplication, PidSort, Primary,
PrimaryExtension,
  //   ProcedureCall, ProcedureCallBody, ProcedureProperty, ProcedureSignature,
RemoteProcedureCall,
  //   RemoteProcedureCallBody, SignalProperty, StateEntryPoint, StateExitPoint,
  //   StateList, StateListToken, Synonym, Syntype, TimerProperty, TypeExpression,
  //   ValueReturningProcedureCall, VariableAccess, VariableProperty
  //
  //   Note 1: 'FormalParameter' and 'ProcedureSignature' cannot be used for root nodes as
they
  //           conflict with 'SortList' (caused by ambiguities in 'behaviourProperty')
  //   Note 2: If one of the tokens is needed in some parts of the AST, a corresponding token
  //           with prefix 'Exc' is used ('Exc' denotes that the token is used exceptionally)

  tokens
  {
    Abstract;                       ActionStatement;                ActualContextParameters;
    ActualParameters;               Adding;                         AgentBody;
    AgentConstraint;                AgentFormalParameters;          AgentInstantiation;
    AgentStructure;                 AgentTypeBody;                  AgentTypeConstraint;
    AgentTypeStructure;             AggregationAggregateEndBoundKind;
    AggregationNotBoundKind;        AggregationPartEndBoundKind;    AggregationTwoEndsBoundKind;
    AlgorithmActionStatement;       AlgorithmAnswerPart;            AlgorithmElsePart;
```

```
    AnchoredSort;                    AnswerPart;                      Any;
    AnyExpression;                   Argument;                        Arguments;
    Assignment;                      AssignmentStatement;             Association;
    AssociationEnd;                  AssociationEndBoundKind;         AssociationNotBoundKind;
    AssociationTwoEndsBoundKind;     AsteriskConnectList;             AsteriskExceptionHandlerList;
    AsteriskExceptionStimulusList;                                    AsteriskInputList;
    AsteriskSaveList;                AsteriskStateList;
    Atleast;                         Block;                           BlockContextParameter;
    BlockDefinition;                 BlockHeading;                    BlockReference;
    BlockType;                       BlockTypeContextParameter;       BlockTypeDefinition;
    BlockTypeHeading;                BlockTypeReference;              BreakStatement;
    ChannelDefinition;               ChannelEndpoint;                 ChannelIdentifiers;
    ChannelPath;                     ChannelToChannelConnection;      ChoiceDefinition;
    ChoiceOfSort;                    ClosedRange;                     CommConDestination;
    CommConTimer;                    Comment;                         CompositeStateBody;
    CompositeStateGraph;             CompositeStateHeading;           CompositeStateReference;
    CompositeStateTypeConstraint;    CompositeStateTypeContextParameter;
    CompositeStateTypeDefinition;    CompositeStateTypeHeading;       CompositeStateTypeReference;
    CompositionCompositeEndBoundKind;                                 CompositionNotBoundKind;
    CompositionPartEndBoundKind;     CompositionTwoEndsBoundKind;     CompoundStatement;
    ConditionalExpression;           ConnectList;                     ConnectPart;
    Constraint;                      ContinuousSignal;                CreateRequest;
    DashNextstate;                   DataTypeDefinition;              DataTypeDefinitionBody;
    DataTypeHeading;                 DataTypeReference;               DataTypeSpecialization;
    Decision;                        DecisionBody;                    DecisionStatement;
    DecisionStatementBody;           Default;                         DefaultInitialization;
    DefinitionSelection;             DefinitionSelectionList;         ElsePart;
    EmptyStatement;                  EnablingCondition;               End;
    EndpointConstraint;              Env;                             ExcCallStatement;
    Exception;
    ExceptionContextParameter;       ExceptionDefinition;             ExceptionDefinitionItem;
    ExceptionHandler;                ExceptionHandlerList;            ExceptionRaise;
    ExceptionStatement;              ExceptionStimulus;               ExceptionStimulusList;
    ExpandedSort;                    Export;                          Exported;
    ExportedAs;                      ExpressionList;                  ExternalChannelIdentifiers;
    ExternalOperationDefinition;     ExternalProcedureDefinition;
ExternalSynonymDefinitionItem;
    Field;                           FieldDefaultInitialization;      FieldsOfSort;
    Finalized;                       FormalContextParameters;         FormalOperationParameters;
    FormalVariableParameters;        FreeAction;                      GateConstraint;
    GateContextParameter;            GateDefinition;                  Handle;
    HandleStatement;                 HistoryDashNextstate;            Identifier;
    IfStatement;                     ImportExpression;                In;
    InOut;                           InnerEntryPoint;                 InputList;
    InputPart;                       Interface;                       InterfaceConstraint;
    InterfaceContextParameter;       InterfaceDefinition;             InterfaceGateDefinition;
    InterfaceHeading;                InterfaceProcedureDefinition;    InterfaceReference;
    InterfaceSpecialization;         InterfaceUseList;                InterfaceVariableDefinition;
    InternalSynonymDefinitionItem;   Join;                            Label;
    LabelledStatement;               LiteralList;                     Literals;
    Local;                           LocalVariablesOfSort;            LoopBreakStatement;
    LoopClause;                      LoopContinueStatement;           LoopStatement;
    LoopStep;                        LoopVariableDefinition;          LoopVariableIndication;
    Method;                          MethodList;                      Methods;
    NameClassLiteral;                NameClassOperation;              NamedNumber;
    Nextstate;                       Nodelay;                         NowExpression;
    NumberOfInstances;               Object;                          Offspring;
    OnException;                     OpAnd;                           OpBinaryMinus;
    OpConcat;                        OpDivide;                        OpEqual;
    OpGreater;                       OpGreaterOrEqual;                OpImplies;
    OpIn;                            OpLess;                          OpLessOrEqual;
    OpMod;                           OpNot;                           OpNotEqual;
    OpOr;                            OpPlus;                          OpRangeCheck;
    OpRem;                           OpStar;                          OpTimes;
    OpUnaryMinus;                    OpXor;                           OpenRange;
    OperationBody;                   OperationDefinition;             OperationHeading;
    OperationParameters;             OperationPreamble;               OperationReference;
```

```
    OperationResult;                OperationSignature;
OperationSignatureInConstraint;
    Operator;                       OperatorList;               Operators;
    Optional;                       Ordered;                    Out;
    OuterEntryPoint;                Output;                     Package;
    PackageDefinition;              PackageHeading;             PackageInterface;
    PackageReference;               PackageUseClause;           ParametersOfSort;
    Parent;                         PartialRegularExpression;   PathItem;
    PriorityInput;                  PriorityInputList;
    Private;                        Procedure;                  ProcedureBody;
    ProcedureConstraint;            ProcedureContextParameter;  ProcedureDefinition;
    ProcedureFormalParameters;      ProcedureHeading;           ProcedurePreamble;
    ProcedureReference;             ProcedureResult;
ProcedureSignatureInConstraint;
    Process;                        ProcessContextParameter;    ProcessDefinition;
    ProcessHeading;                 ProcessReference;           ProcessType;
    ProcessTypeContextParameter;    ProcessTypeDefinition;      ProcessTypeHeading;
    ProcessTypeReference;           Protected;                  Public;
    Qualifier;                      Raise;                      RaiseStatement;
    Raises;                         RangeCondition;             Redefined;
    ReferenceSort;                  RegularInterval;            Remote;
    RemoteProcedure;                RemoteProcedureContextParameter;
    RemoteProcedureDefinition;      RemoteProcedureReject;
RemoteVariableContextParameter;
    RemoteVariableDefinition;       RenamePair;                 Renaming;
    Reset;                          ResetClause;                Result;
    Return;                         ReturnStatement;            SDLSpecification;
    SavePart;                       SelectDefinition;           Self;
    Sender;                         Set;                        SetClause;
    Signal;                         SignalConstraint;           SignalContextParameter;
    SignalDefinition;               SignalDefinitionItem;       Signallist;
    SignalList;                     SignalListDefinition;       SignalListItem;
    SignalReference;                SizeConstraint;             SomeExpression;
    SomePrimaryExtension;           SomeProcedureCall;
    SomeSort;                       SortConstraint;             SortContextParameter;
    SortList;                       SortSignature;              Specialization;
    SpellingTerm;                   SpontaneousTransition;      Start;
    State;                          StateAggregation;           StateAggregationHeading;
    StateAggregationTypeDefinition;                             StateAggregationTypeHeading;
    StateConnectionPoints;          StateEntryPoints;           StateExitPoints;
    StateExpression;                StatePartitionConnection;   StateType;
    Stimulus;                       Stop;                       StructureDefinition;
    StructurePrimary;               Synonym;                    SynonymContextParameter;
    SynonymDefinition;              SyntypeDefinition;          System;
    SystemDefinition;               SystemHeading;              SystemType;
    SystemTypeDefinition;           SystemTypeHeading;          SystemTypeReference;
    Task;                           TerminatorStatement;        This;
    Timer;                          TimerActiveExpression;      TimerContextParameter;
    TimerDefaultInitialization;     TimerDefinition;            TimerDefinitionItem;
    Transition;                     TransitionOption;           Type;
    TypeReferenceProperties;        TypebasedBlockDefinition;   TypebasedCompositeState;
    TypebasedProcessDefinition;     TypebasedStatePartitionDefinition;
    TypebasedSystemDefinition;      ValidInputSignalSet;        Value;
    Variable;                       VariableContextParameter;   VariableDefinition;
    VariableDefinitionStatement;    VariableField;              VariableIndex;
    VariablesOfSort;                ViaGate;                    ViaPath;
    Virtual;                        VirtualityConstraint;       Void;
  }

{
  public:

    unsigned int numberOfErrors() const;
    void resetErrors();

    static const string &versionInfo();
```

```
  | start04:BLOCK!          { ## = #[Block];       SETLINECOLUMN( ##, start04 ); }
  | start05:BLOCK! TYPE!    { ## = #[BlockType];   SETLINECOLUMN( ##, start05 ); }
  | start06:PROCESS!        { ## = #[Process];     SETLINECOLUMN( ##, start06 ); }
  | start07:PROCESS! TYPE!  { ## = #[ProcessType]; SETLINECOLUMN( ##, start07 ); }
  | start08:STATE!          { ## = #[State];       SETLINECOLUMN( ##, start08 ); }
  | start09:STATE! TYPE!    { ## = #[StateType];   SETLINECOLUMN( ##, start09 ); }
  | start10:PROCEDURE!      { ## = #[Procedure];   SETLINECOLUMN( ##, start10 ); }
  | start11:SIGNAL!         { ## = #[Signal];      SETLINECOLUMN( ##, start11 ); }
  | start12:TYPE!           { ## = #[Type];        SETLINECOLUMN( ##, start12 ); }
  | start13:OPERATOR!       { ## = #[Operator];    SETLINECOLUMN( ##, start13 ); }
  | start14:METHOD!         { ## = #[Method];      SETLINECOLUMN( ##, start14 ); }
  | start15:INTERFACE!      { ## = #[Interface];   SETLINECOLUMN( ##, start15 ); }

  ;

/* 6.4 Informal text, page 23 */

informalText :  //  no node type change due to ambiguity in various rules

  CharacterString

  ;

/* 6.7 Comment, page 24 */

end { bool cmt = false; } :

  ( comment { cmt = true; } )? start:Semicolon!

  { ## = #( [End], ## ); if ( !cmt ) SETLINECOLUMN( ##, start ); }

  ;

comment! :

  start:COMMENT! c:CharacterString

  { c -> setType( Comment ); ## = #c; SETLINECOLUMN( ##, start ); }

  ;

/* 7.1 Framework, page 25 */

sdlSpecification :

  (   ( ( packageUseClause )* PACKAGE ) =>
      package
    | systemSpecification )
  // ( package )*     package is also included in referencedDefinition
  ( referencedDefinition )*
  EOF!

  { ## = #( [SDLSpecification], ## ); }

  ;

systemSpecification :

  textualSystemSpecification

  ;

package :

  packageDefinition

  ;
```

```
textualSystemSpecification :

    (   SYSTEM typebasedSystemHeading | BLOCK typebasedBlockHeading
      | PROCESS typebasedProcessHeading ) =>
    textualTypebasedAgentDefinition
  | agentDefinition

  ;

/* 7.2 Package, page 26 */

packageDefinition :

  ( packageUseClause )*
  packageHeading end
  ( entityInPackage )*
  ENDPACKAGE! ( /*package*/ Name! )? end

  { ## = #( [PackageDefinition], ## ); }

  ;

packageHeading :

  start:PACKAGE! ( qualifier )? nameDef[ SDLName::package ]
  ( packageInterface )?

  { ## = #( [PackageHeading], ## ); SETLINECOLUMN( ##, start ); }

  ;

entityInPackage :

    ( agentTypeReference ) =>
    agentTypeReference
  | ( packageReference ) =>
    packageReference
  | ( signalReference ) =>
    signalReference
  | ( dataTypeReference ) =>
    dataTypeReference
  | ( procedureReference ) =>
    procedureReference
  | ( compositeStateTypeReference ) =>
    compositeStateTypeReference
  | ( interfaceReference ) =>
    interfaceReference
  | ( ( packageUseClause )* ( SYSTEM TYPE | typePreamble ( BLOCK TYPE | PROCESS TYPE ) ) ) =>
    agentTypeDefinition
  | ( ( packageUseClause )* PACKAGE ) =>
    packageDefinition
  | ( typePreamble SIGNAL ) =>
    signalDefinition
  | signalListDefinition
  | remoteVariableDefinition
  | ( typePreamble ( VALUE | OBJECT ) TYPE | ( virtuality )? INTERFACE | SYNTYPE | SYNONYM )
=>
    dataDefinition
  | ( PROCEDURE | ( packageUseClause )* procedureHeading ) =>
    procedureDefinition
  | remoteProcedureDefinition
  | compositeStateTypeDefinition
  | exceptionDefinition
  | selectDefinition
  | macroDefinition
  | association
```

```
  ;

definition :

    ( ( packageUseClause )* PACKAGE ) =>
    packageDefinition
  | ( ( packageUseClause )* ( SYSTEM TYPE | typePreamble ( BLOCK TYPE | PROCESS TYPE ) ) ) =>
    agentTypeDefinition
  | ( ( packageUseClause )* ( SYSTEM | BLOCK | PROCESS ) ) =>
    agentDefinition
  | ( ( packageUseClause )* ( virtuality )? STATE ( AGGREGATION )? TYPE ) =>
    compositeStateTypeDefinition
  | ( ( packageUseClause )* ( compositeStateHeading | stateAggregationHeading ) ) =>
    compositeState
  | ( PROCEDURE | ( packageUseClause )* procedureHeading ) =>
    procedureDefinition
  | operationDefinition
  | macroDefinition

  ;

/* 8.1.1.1 Agent types, page 31 */

agentTypeDefinition :

    ( ( packageUseClause )* SYSTEM ) =>
    systemTypeDefinition
  | ( ( packageUseClause )* typePreamble BLOCK ) =>
    blockTypeDefinition
  | processTypeDefinition

  ;

agentTypeStructure { bool atBody; } :

  ( validInputSignalSet )?
  atBody = loopstar_in_agentTypeStructure
  (   (   STATE ( SUBSTRUCTURE | AGGREGATION )
        | ( packageUseClause )* ( compositeStateHeading | stateAggregationHeading )
          end SUBSTRUCTURE ) =>
      ( statePartitioning )?
    | agentTypeBody { /* use value of 'atBody' for checks and AST construction */ } )

  { ## = #( [AgentTypeStructure], ## ); }

  ;

loopstar_in_agentTypeStructure returns [ bool atBody ] { bool loopVar; } :
        .               // 'entityInAgent' moved to bottom due to its large number of
alternatives

    (   ENDSYSTEM | ENDBLOCK | ENDPROCESS
      | ONEXCEPTION | START | ( state ) => state /*!*/ | EXCEPTIONHANDLER | CONNECTION
      | STATE ( SUBSTRUCTURE | AGGREGATION )
      | ( packageUseClause )* ( compositeStateHeading | stateAggregationHeading )
        end SUBSTRUCTURE ) =>
    { atBody = true; }
    // end of loop
  | (   ( agentReference ) =>
        agentReference                          { atBody = false; }
      | channelDefinition                       { atBody = false; }
      | gateInDefinition                        { atBody = true; }
      | (   SYSTEM typebasedSystemHeading | BLOCK typebasedBlockHeading
          | PROCESS typebasedProcessHeading ) =>
        textualTypebasedAgentDefinition         { atBody = false; }
      | ( ( packageUseClause )* ( systemHeading | blockHeading | processHeading ) ) =>
```

```
    agentDefinition                         { atBody = false; }
  | entityInAgent                           { atBody = true; } )
loopVar = loopstar_in_agentTypeStructure

{ atBody = atBody && loopVar; }


ePreamble returns [ bool preamble = false ] :

    virtuality { preamble = true; }
  | abstract   { preamble = true; } )?


entTypeAdditionalHeading :    // no root node because 'agentTypeAdditionalHeading'
                              // may derive to the empty string
( formalContextParameters )? ( virtualityConstraint )?
agentAdditionalHeading

;

entTypeReference :

  systemTypeReference
| ( typePreamble BLOCK ) =>
  blockTypeReference
| processTypeReference

;

gentTypeBody :

( ( onException )? start )?
( state | exceptionHandler | freeAction )*

{ ## = #( [AgentTypeBody], ## ); }

;

* 8.1.1.2 System type, page 33 */

ystemTypeDefinition :

( packageUseClause )*
systemTypeHeading end agentTypeStructure
ENDSYSTEM! TYPE! ( ( qualifier! )? /*systemType*/ Name! )? end

{ ## = #( [SystemTypeDefinition], ## ); }

;

ystemTypeHeading :

start:SYSTEM! TYPE! ( qualifier )? nameDef[ SDLName::systemType ]
agentTypeAdditionalHeading

{ ## = #( [SystemTypeHeading], ## ); SETLINECOLUMN( ##, start ); }

;

systemTypeReference :

start:SYSTEM! TYPE! /*systemType*/ identifier typeReferenceProperties

{ ## = #( [SystemTypeReference], ## ); SETLINECOLUMN( ##, start ); }
```