# Interpretational Approach to Service Logic Execution

Ivaylo I. Atanasov[1] and Evelina N. Pencheva[2]

***Abstract* – The paper presents general aspects of implementation of a new mark-up language used for service creation. The language interpreter consists of two parts: lexical analyzer which recognizes the lexical units of the language, and the parser which performs syntax analysis of the input sequence of tokens in order to determine its grammatical structure with respect to language formal grammar. During interpretation, for each language construction correctly recognized by the parser, an appropriate Java construction is activated.**

***Keywords* – Mark-up language for service creation, lexical analysis, parsing, interpreting**

## I. INTRODUCTION

Markup languages have found their way into almost every facet of telecommunications from provisioning services through network management systems and even scripting languages for automated voice services.

Because of fast and complicated hardware there is no need to optimize the use and encoding of information in protocols and databases in binary form. The simplicity and ease of understanding make text encoding preferable both in representing information in databases and in encoding protocol messages. This, combined with the view that content means revenue, is giving rise of markup languages for next generation service creation [1], [2].

There exist several markup languages proposed for service creation. Languages like CPL (Call processing Language), VoiceXML (Voice eXtensible Markup Language), CCXML (Call Control eXtensible Markup Language) are mainly oriented towards call control i.e. the creation, manipulation, termination and teardown of communication sessions. None of the existing markup languages supports functions like mobility, user status, terminal capabilities, charging and others. These network functions can be accessed by Parlay/OSA (Open Service Access) interfaces. Parlay/OSA is a service framework which defines application programming interfaces (APIs) for third party service developers. The idea behind API is to hide network specifics and protocol complexity for service developers.

We suggest a new mark-up language called **S**ervice **L**ogic **P**rocessing **L**anguage (SLPL) which is developed to meet challenges of service creation [3]. The language supports the whole palette of network functions exposed by Parlay/OSA interfaces.

One of the factors that determine language usability depends on availability of supporting tools. An SLPL interpreter is developed that makes lexical analysis of service logic description, builds an abstract syntax tree and makes mapping of abstract tree's nodes to appropriate Java constructions. To ease the process of service description a translator from IDL (Interface Description Language) to SLPL is also developed. The IDL is a language used to specify data types and method definitions of Parlay/OSA APIs. Because of the huge amount of data types and methods defined, this tool allows translation of Parlay/OSA definitions into SLPL ones.

In the paper we present some aspects of the implementation of SLPL interpreter and of translator from IDL to SLPL.

## II. SLPL INTERPRETATIONAL APPROACH

The interpretation of the logic is separated into four processing phases – front processing, lexical analysis, syntactical analysis, and execution. Fig.1 shows the interpretational approach of execution of service logic described in SLPL.

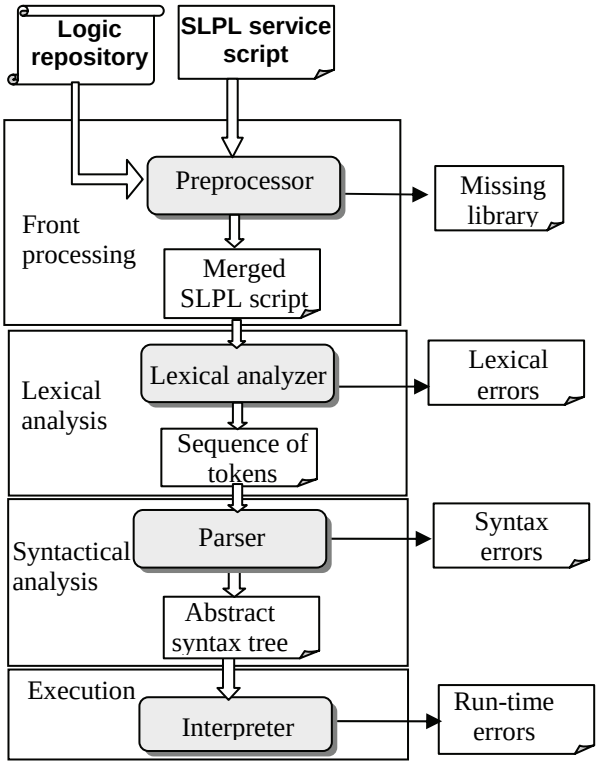


Fig.1 Interpretational approach

The front processing is in charge of loading the service script i.e. making an instance of the logic script. Ready-to-use parts of script containing SLPL data types and methods definitions are imported from the SLPL library. After

[1]Ivaylo I. Atanasov is with the Faculty of Communications and Communication Technologies, Technical University of Sofia, 1000 Sofia, Bulgaria, E-mail: iia@tu-sofia.bg

[2]Evelina N. Pencheva is with the Faculty of Communications and Communication Technologies, Technical University of Sofia, 1000 Sofia, Bulgaria, E-mail: enp@tu-sofia.bg

including definitions in the original instance, the SLPL preprocessor produces a merged, extended instance.

The main task of the SLPL lexical analyzer is to recognize the lexical units of the language like identifiers, key words, literals and so on. The extended instance of service script is lexically converted into a sequence of tokens and then the sequence is passed as input to the parser.

The SLPL parser performs syntax analysis of the input sequence of tokens in order to determine its grammatical structure with respect to SLPL formal grammar. The SLPL parser is to decide whether the sequence is acceptable in the terms of the syntactic rules of the language. If it is to be rejected, then error log is open. Parsing transforms input sequence of tokens into a data structure (an abstract tree), which is suitable for later processing and which captures the implied hierarchy of the input.

During real processing for each language construction recognized correctly by the SLPL parser, an appropriate Java construction is activated.

## III. SLPL INTERPRETER

The main task of SLPL lexical analyzer is to break SLPL service logic description into tokens. The lexical analyzer processes the input sequence of characters to recognize the lexemes and to evaluate their type.

The input sequence scanning is based on finite state machine. It has encoded within it information on the possible sequences of characters that can be contained within any of the tokens it handles.

The finite state machine (FSM) of the lexical analyzer is table-based. The state-table defines all details of the behavior of FSM. It consists of three columns: in the first column state names are used, in the second the virtual conditions built out of input names using the positive logic algebra are placed and in the third column the output names appear.

Fig.2 shows an example of FSM that recognizes the key words *val*, *valref* and *value*.

The lexeme types in SLPL are the following:

- literals (integer, long, float, double, octet, logical and character)
- operators (for example equation)
- separators (for example space, slash, angular brackets, double quotes)
- identifiers
- key words.

In order to construct a token, the lexical analyzer needs to evaluate the characters of the lexeme that is to produce a value. The lexeme's type combined with its value is what properly constitutes a token, which can be given to a parser.

For example, let us consider the SLPL language construction for synchronization. For the input

<wait timeout="20"/>

the SLPL lexer ignoring spaces recognizes the type and value of the tokens listed in Table 1.

The sequence of tokens is passed as input to the SLPL parser.

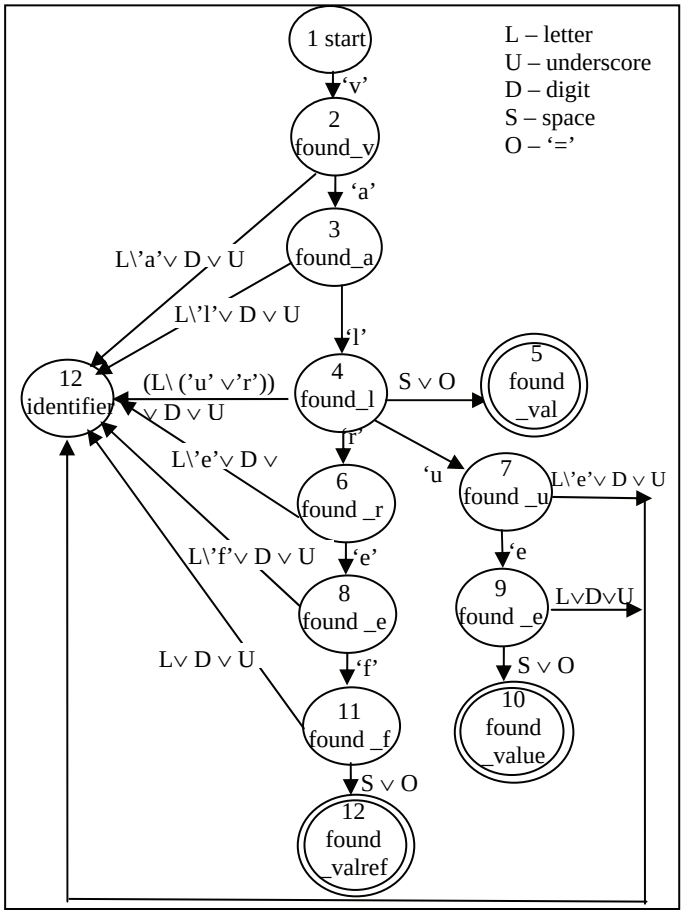


Fig.2 FSM recognizing SLPL lexemes

TABLE 1
LEXEMES AND THEIR TYPE

| | |
|---|---|
| < | separator |
| wait | key word |
| timeout | key word |
| = | operator |
| " | separator |
| 20 | integer literal |
| " | separator |
| / | separator |
| > | separator |

The SLPL parser analyzes the sequence of tokens to determine its grammatical structure with respect to the SLPL formal grammar. Its task is essentially to determine if and how the input can be derived from the start symbol within the rules of the SLPL formal grammar. This is done in a top-down parsing manner. The SLPL starts with the start symbol and tries to transform it to the input. The parser starts from the largest elements and breaks them down into incrementally smaller parts. It is LL parser i.e. it parses the input from left to right, and constructs a leftmost derivation of the sentence.

The parser builds abstract syntax tree, which is a finite, labeled, directed tree, where the internal nodes are labeled by grammar rules, and the leaf nodes represent the terminal or nonterminal symbols.

The SLPL grammar is formally described in Augment Backus Naur Form (ABNF). For example, the language construction for synchronization “wait-statement” used to wait for result of network provided service is formally defined with the grammar rules shown in Fig. 3.

```
1.  wait_def        = LB "wait" [timeouted](slash GB/ GB
                      event_list); wait for an even or timeout
2.  timeouted       = "timeout” is DQUOTE integer-val DQUOTE
3.  event_list      = 1*event_list_item LB slash "wait" GB
4.  event_list_item = LB“event” a_name slash GB
5.  a_name          = "name" is DQUOTE simple_name DQUOTE
6.  integer_literal = 1*DIGIT
7.  simple_name     = ALPHA*(ALPHA/ DIGIT/ underscore)
8.  slash           = %d47; ‘/’
9.  is              = %d61; ‘=’
10. underscore      = %d95; ‘_’
11. LB              = %d60; ‘<’
12. GB              = %d62; ‘>’
13. ALPHA           = %d65-90 / %d97-122; A-Z a-z
14. DIGIT           = %d48-57: 0-9
```

Fig.3 Grammar rules for definition of wait-statement

The SLPL parser has an input buffer, a stack on which it keeps (terminal and nonterminal) symbols from the grammar, a parsing table which tells what grammar rule to use given the symbols on top of its stack and its input type.

In order to construct the parsing table, that is to establish what grammar rule the parser should choose if it has a nonterminal *A* on the top of its stack and a symbol *a* on its input stream, we have used the algorithm described in [4].

For example, Table 2 shows the parsing table for the grammar rules defining the “wait-statement”.

TABLE 2
PARSING TABLE FOR WAIT-STATEMENT

| | $ | wait | timeout | event | name |
|---|---|---|---|---|---|
| wait_def | | | | | |
| wait_def_next | | | 3 | | |
| wait_def_ext | | | | | |
| timeouted | | | 6 | | |
| event_list | | | | | |
| event_list_ext | | | | | |
| event_list_item | | | | 10 | |
| a_name | | | | | 11 |

TABLE 2
PARSING TABLE FOR WAIT-STATEMENT
(CONTINUE)

| | slash | GB | is | DQUOTE | integer_ literal | simple _name |
|---|---|---|---|---|---|---|
| wait_def | | | | | | |
| wait_def_next | 2 | 2 | | | | |
| wait_def_ext | 4 | 5 | | | | |
| timeouted | | | | | | |
| event_list | | | | | | |
| event_list_ext | 9 | | | | | |
| event_list_item | | | | | | |
| a_name | | | | | | |

The terminal ‘$’ indicates the bottom of the stack. The resulting abstract tree from the input sequence of tokens for

<wait timeout="20"/>

is shown in Fig. 4

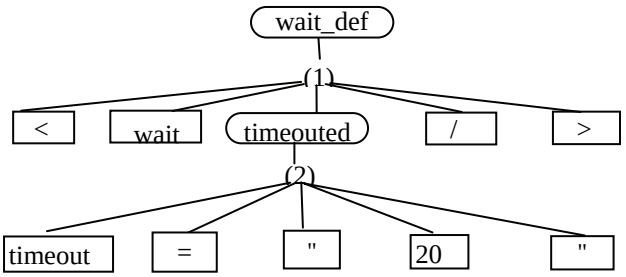


Fig. 4 Abstract syntax tree built for ‘wait’-statement

The overview of the abstract syntax tree generated from an SLPL service logic description is shown in Fig. 5.

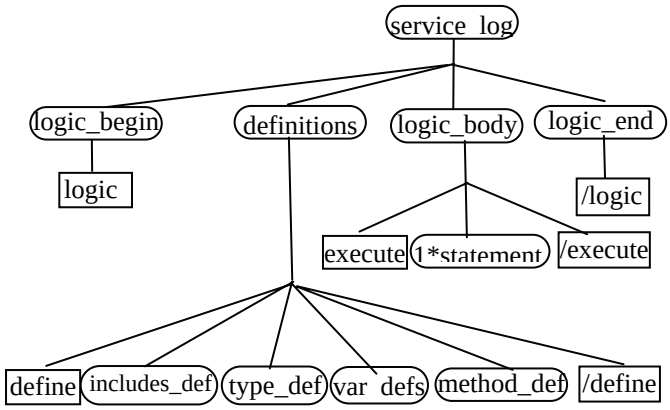


Fig.5 Overview of syntax tree of SLPL grammar description

During processing correct language structures are compared with templates of semantic descriptions and converted into Java calls.

For example, the SLPL language construction for method invocation with capturing exceptions is shown in Fig. 6 and the corresponding Java code activated is shown in Fig. 7.

```
<try>
   <invoke>
      <method name = "locationReportReq">
         <arguments>
            <argument name = "application" valref = "theAppl"/>
            <argument name = "users" valref = "theUsers"/>
         </arguments>
      </method>
   </invoke>
   <catch>
      <exception name = "P_APPLICATION_NOT_ACTIVATED"/>
      <exit/>
   </catch>
</try>
```

Fig. 6 SLPL construction for method invocation

```
try {locationReportReq( theAppl, theUsers );
    } catch ( ApplicationNotActivatedException e )
   { exit(); }
```

Fig. 7 An example of Java code corresponding to method invocation

## IV. TRANSLATION FROM IDL TO SLPL

The Parlay/OSA interfaces are specified in IDL which is programming language independent. A huge amount of data types on which methods operate are included in IDL specification. To reduce efforts needed for data types and method definition in SLPL an "include-statement" is provided. This construction allows including SLPL descriptions of methods and data types in the definition part of the service logic script. These ready-to-use parts of script are located in the script repository, accessed in shared library manner.

To create SLPL script repository a translator form IDL to SLPL is developed. As it is shown in Fig. 8, the translator is supplied with IDL description as input and the result is SLPL description as output.

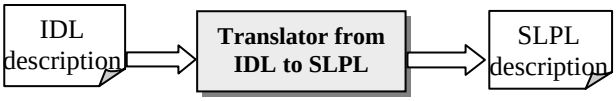


Fig. 8 Translator from IDL to SLPL

The formal IDL definition of an interface includes data types and methods. Each method has a type (the result type), a list of arguments of certain type, and a list of exceptions that it can raise.

The translator makes lexical analysis of an IDL description to determine tokens and then performs syntactical analysis to build an abstract syntax tree corresponding to IDL description. During code generation the translator converts the syntactically correct IDL descriptions to SLPL ones.

Fig.9 shows a part of SLPL grammar rules that formally define an interface with data types and methods.

## V. CONCLUSION

The suggested markup language SLPL possesses greater expressive power in comparison with existing markup languages for service creation. The language supports all network function exposed by Parlay/OSA interface. Provided language construction for flow control brings SLPL near to programming languages.

The language usability is provided by development of language supporting tools such as SLPL interpreter and translator from IDL to SLPL. These tools are written in Java to achieve portability. Another reason for Java choice as implementation language is that we use Ericsson Network Resource Gateway SDK (version R5A02) that simulates Parlay/OSA interfaces to verify the SLPL functional capabilities. The interface method calls of Parlay/OSA interface simulator are in form of Java code.

The approach enables rapid prototyping, rapid application development, and easy end-user customization. These facts free developers to focus on the creation of new types of applications that may bring new revenues to service providers.

1. interface_def = LB "interface" a_name GB interface_body LB slash "interface" GB
2. interface_body = inheritance type_defs method_defs
3. inheritance = LB “inherits” GB inherit_instances LB slash “inherits” GB
4. inherit_instance= 1*(LB “interface” a_name slash GB)
5. type_defs = (LB "types" 1*type_def slash "types" GB) / (LB "types" slash GB)
6. type_def = simple_type / complex_type / alias_type
7. simple_type = "integer" / "long" / "float" / "double" / "boolean" / "string" / "char" / "octet" / "reference" / "Timer" / "Time" / "Date" / "DateAndTime" / "Duration“
8. complex_type = enumerated_type / structured_type / sequence_type / union_type
9. method_defs = (LB "methods" slash GB) / methods_def
10. methods_def = LB "methods" GB 1*method_def LB slash "methods" GB
11. method_def = LB “method” a_name method_args [body_def] return_type [exceptions_raised] LB slash "method" GB
12. method_args = LB "arguments" GB 1*method_arg LB slash "arguments" GB
13. method_arg = LB "argument" a_name a_type slash "argument" GB
14. return_type = (LB "returns" slash GB) / return_def
15. return_def = LB "returns" a_type slash GB
16. exceptions_raised= LB "raises" 1*exception_def slash "raises" GB
17. exception_def = LB "exception" (exception_by_type / exception_by_name) slash "exception" GB
18. exception_by_type= a_type
19. exception_by_name= a_name
20. body_def = LB "body" GB 1*statements LB slash “body“ GB
21. statements = assign_statement / invoke_statement / wait_statement / if_statement / case_statement / while_statement / goto_statement / try_statement / arithmetic_operations / new_statement / exit_statement

Fig.9 Formal grammar tool for interface definition in SLPL

## REFERENCES

[1] Bakker, J. Tweedie, D. and M. Umnehopa, “Evolving service Creation; New developments in Network Intelligence”, http://www.argreenhouse.com/papers/jlbakker/Bakker-elenor.pdf

[2] Bakker J-L, R. Jain, “Next Generation Service Creation Using XML Scripting Languages”, http://www.argreenhouse.com/papers/jlbakker/bakker-icc2002.pdf

[3] Atanasov I., E. Pencheva, “A Mark-up Approach to Add Value”, IJIT Enformatika, vol.3, Number 4, pp 267-276

[4] Aho, Sethi, Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[5] Common Object Request Broker Architecture (CORBA), v3.0, *OMG IDL Syntax and Semantics*, http://www.omg.org/docs/formal/00-10-07.pdf

[6] Ericsson Network Resource Gateway SDK (version R5A02) http://www.ericsson.com/mobilityworld/sub/open/technologies/parlay/tools/parlay_sdk