An Approach to Provide Network Capabilities-based Added Value

I. Atanasov, E. Pencheva

Key Words: *OSA interfaces; XML-based languages; NGN service creation.*

Abstract. *A new mark-up approach to next generation service creation is proposed. The approach can be summarized as development of a scripting language SLPL that exploits OSA (Open Service Access) service capability features like mobility, call control, data session control, messaging, user interaction etc. Language constructions for data types and method definitions, flow control, time measuring and supervision, database access are provided. Examples of SLPL descriptions are presented.*

I. Introduction

Next Generation Network (NGN) shall provide the capabi–lities (infrastructure, protocols, etc.) to make creation, deploy–ment and management of all kinds of services (known or not yet known) possible. This comprises a wide variety of services including voice, video, audio and visual data, via session and interactive based services in unicast, multicast and broadcast modes. Wireline and wireless technologies can be used inter–changeably for delivery of services. Furthermore, within the NGN there is an increased emphasis on service customization by service providers whereby some of them will offer their customers the possibility to customize their own services. NGN should be comprised of service related APIs (Application Prog–ramming Interfaces) in order to support the creation, provision–ing and management of services.

The most promising umbrella technology for service delivery in NGN is Parlay/Open Service Access (OSA). OSA covers existing technologies as service capabilities and provides for them features and services. The way OSA is defined, it can easily take onboard new technologies.

One of the ways of implementing OSA is by the use of eXtensible Mark-up Language (XML). The beauty of XML is that it is both human and machine-readable. It is easy to write a parsing program to recognize XML tags, to find them in an XML message and then to extract the information contained in the tagged fields.

In fact, the use of XML has become prevalent in general and there exist a number of XML-based languages that support NGN service creation.

XML-based languages like CPL, VoiceXML, SCML [1,2] possess restricted expressive power related to call control and call-related user interactions. These languages are mainly used to model and execute initiation, manipulation, termination and clearing of communication sessions.

Some of existing XML-based languages for NGN service creation are platform dependent meaning their orientation toward SIP, Java virtual machine and others, while none of them takes into account the OSA technology [3,4]. OSA APIs provide much more network capabilities than call and data session control. Using functional abstraction provided by OSA APIs the service developer can benefit added value from network capabilities such as mobility, charging, presence and availability, user interaction and others. OSA APIs hide underlying network and protocol complexity from service developers and make easier service creation for wider part of IT community.

The ongoing research concentrates on a new mark-up approach to service creation. It is oriented towards OSA APIs in order to exploit service capability features like mobility, call control, data session control, messaging, user interaction etc. To approve the approach applicability we define XML-based language constructions that support the whole variety of OSA APIs and allow description of service logic in terms of flow control, time measuring and supervision and database access.

In this paper we present in brief the methodology of deriving language constructions that correspond to application domain. To illustrate the idea of language synthesis we present language constructions for data type definition and give examples.

II. Methodology of Approach to Service Creation

The methodology of the approach for the NGN service creation might be depicted as development of a language that contains domain specific constructions. The language has to reflect the objects and their relationships in the application domain under given constraints. The constraints are determined by support of access to network functions through OSA APIs. OSA APIs provide programmability of network resources in terms of objects and methods, data types and parameters that operate on those objects. Consequently, the approach has to provide language constructions for invocation of the OSA API methods and processing the results returned in the context of service logic. To allow time-dependent processing and service customization, constructions for time handling and database access are needed. The methodology for deriving added value from network functions by tagging is shown in *figure 1*.

Based on the domain analysis it can be stated that language constructions should provide means for the following:

- Description of data types, supported by OSA interfaces and definition of variables of supported data types (statements for data

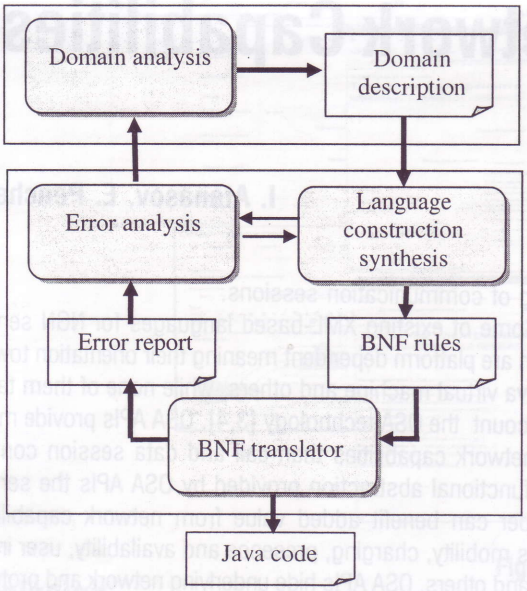

Figure 1. Methodology of the markup approach to service creation

types and variable definitions).

• Description of OSA interface and application methods with parameters, type of result returned and exceptions that might arise (statements for interface and method definitions).

• Invocation of OSA interface methods to access network functions ('invoke-statement').

• Capturing exceptions to define exception processing ('raises-statement').

• Method synchronization to provide synchronous commu–nications ('wait-statement').

• Flow control to provide decision making, multiple choice, action reiteration etc. ('if-statement', while-statement', 'case-statement').

• Elementary arithmetic operations to allow applying of simple algorithms (addition, subtraction, multiplication, division).

• Tools for time measuring and supervision to allow time-based decision making (timers, operation to extract time and date).

• Database access to allow retrieve, update, insert and delete data in external databases (methods for data retrieve, update and conversion).

We call the language SLPL — Service logic processing language. SLPL is an XML- based language and proposes declarative way for service logic description.

The synthesis of language constructions comes to define the syntactic rules and to clarify the semantics of the rules. At this phase it should produce the language rules in form of alphabet, sets of keywords, separators, operators, literals, grammar productions and code mapping. The textual form is converted into more precise formal definition — Backus-Naur forms (BNF). In case of ambiguity or inconsistency the error analysis is initiated.

Using the previously defined code mappings and the syntactic rules translation into Java code is performed in order to assist the further building of language translator/interpreter. An error report is generated on having syntactic inconsistency or incorrect format of BNFs.

Two reasons determine the choice of Java as the implementation language. The first one is a that Java is a platform independent language. Java programs are not compiled directly to machine code, as in case with C++ and most other programming languages, but are translated to an intermediary byte code for a virtual machine.

Another reason for the choice of Java is that the applicability of the SLPL and the suggested mark-up approach for service creation is verified by a Java based Parlay/OSA simulator [7]. The Ericsson Network Resource Gateway SDK (version R5A02) simulates Parlay/OSA interfaces and its interface method calls are Java-based.

The BNF translator can be considered as a 'translator-generator'. If we pass new BNF rules at the entrance of this translator-generator it will generate Java code that 'understands' these grammar rules.

The BNF translator can find some syntactic or semantic error like incompleteness or ambiguity in the BNF rules, defining SLPL formal grammar. Ranking the severity of the errors reported it might be necessary to reconsider the domain description, running again a new iteration of domain analysis. This might be due to lack of certain rules i.e. semantic inconsistency. On the other hand, the language analysis might need a rerun in order to reformulate the BNF rules found as erroneous.

The approach is assuming that the service logic execution environment is the 'natural container' for the execution phase of the service logic script in its life-cycle.

The interpretation of the logic is separated into two processing phases — front one and back one. *Figure 2* shows the interpretational approach of the SLPL service execution.

The front processing phase is in charge of loading the service script i.e. making an instance of the logic script.

The OSA interfaces are specified in Interface Description Language (IDL) which is programming language independent. An IDL specification describes an object interface in terms of the methods it supports. Each method has a type (the result type), a list of parameters and a list of exceptions that it can generate. 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 'import' construction is provided which allows including SLPL descriptions of methods and data types in the definition part of the service logic script. After merging some 'ready-to-use' parts of script, which might be located in the script repository, accessed in shared library manner, and thus doing the preprocessing over 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, terminal symbols, literals and so on. The extended instance of the service script is lexically converted into a sequence of tokens and then the sequence is passed as an 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 which is omitted from the figure for the sake

of simplicity. Parsing transforms input sequence of tokens into a data structure (a tree), which is suitable for later processing and which captures the implied hierarchy of the input.

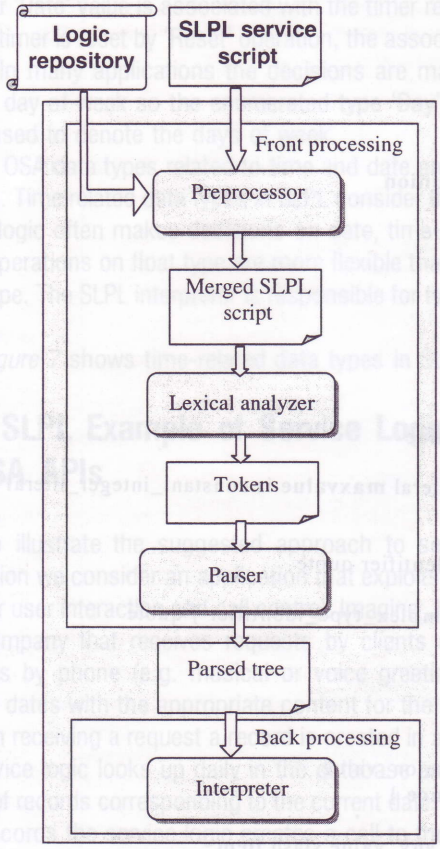

Figure 2. Interpretational approach

The parsed tree of service logic is the internal format which is then the input for the interpretation. The interpreter is doing the mapping of tokens' semantics onto the Java code which has to be invoked. However, if any kind of run-time exceptional situation arises, it is firstly logged and just then caught for further consumption.

To demonstrate the language synthesis we provide markup constructions for data type definition in the next section.

III. Data Type Definition in SLPL

The structure of a SLPL service script is modular and encompasses definition part and executive part. The definition part involves type definitions, variable definitions and definition of the methods supported by the application-side interfaces. The executive part is built mainly of invocations of methods sup–ported by the server-side interfaces.

Here we put the stress on the definition part where the local types, variables and methods are defined. The section considers SLPL constructions for data type definitions according to data types defined for OSA APIs.

General data type definitions are specified in 3GPP TS 29.198-2 Recommendation and specific APIs data type definitions are given in the corresponding 3GPP TS 29.198-xx Recommendations.

The types defined are simple or complex types. Simple types encompass the following: Integer, Long, String, Float, Double and Boolean. Complex types are specified as Enumerated types, Sequences of data elements, Numbered sets of data elements and Tagged choices of data elements. The formal SLPL grammar for OSA APIs data types in Backus-Naur Form (BNF) is shown in *Figure 3*. The Language terminal symbols are presented in bold.

The SLPL definition of an enumerated type is denoted by the terminal symbol 'structure' and consists of a structure name and an enumerated type body. The terminal symbol 'o' (like 'only_one_of') is used to enumerate the elements in the enumerated type body. This definition encompasses the productions 4, 5 and 6.

The SLPL definition of a sequence of data elements is also denoted by the terminal symbol 'structure' but consists of a structure name, structure type and structure elements. This definition encompasses the productions 7, 8 and 9.

The SLPL definition of a numbered set of data elements is denoted by the terminal symbol 'sequence' and consists of a sequence name and sequence type (productions 10 and 11).

The SLPL definition of a tagged choice of data elements is presented with productions from 12 to 21. It is denoted by the terminal symbol 'union' and consists of a union name, a switch node and a case list. The switch node (denoted by 'switch') defines the tag element type used for the choice, while the case list enumerates the possible data elements based on the tagged choice. The terminal symbols 'on val' are used to denote each of the choices. The definition encompasses the default choice also.

The rest productions from 22 to 33 in the *figure 3* are more general and support the enlisted non-terminal symbols

The declaration of variables of data types defined follows the similar approach. For example the productions 34, 35, 36, 37 and 38 in *figure 3* present rules for SLPL declaration of a variable of numbered set of data element type. The terminal symbol 'item' denotes a data element, while the terminal symbol 'index' points the consecutive number of that data element.

To give an example let us consider the SLPL definition of a tagged choice. According to Recommendations TpAoCOrder is a tagged choice of data elements that specifies the charge plan for the call. The tag element type used for the choice is TpCallAoCOrderCategory. When the tag element has value P_CHARGE_ADVICE_INFO, then the union has an element of name ChargeAdviceInfo which is of type TpChargeAdviceInfo. When the tag element has value P_CHARGE_PER_TIME, then the union has an element of name ChargePerTime which is of type TpChargePerTime. When the tag element has value P_CHARGE_NETWORK, then the union has an element of name NetworkCharge which is of type TpString.

Figure 4 shows an example of a SLPL description of a tagged choice.

As an example of a SLPL variable declaration let us consider the method periodicLocationReportingStartReq() of Mobility OSA API. The method is used to request periodic reports of the location for several users. Before the method invocation

```
1.  type_spec              ::=  simple_type | complex_type
2.  simple_type            ::=  integer | long | float | double | boolean | string
3.  complex_type           ::=  enumerated_type | structured_type | sequence_type | union_type
4.  enumerated_type        ::=  structure t_name enum_body slash structure
5.  enum_body              ::=  option enum_elements slash option
6.  enum_elements          ::=  constant_element_spec+
7.  structured_type        ::=  structure t_name { slash | struct_type }
8.  struct_type            ::=  struct_elements slash structure
9.  struct_elements        ::=  element_spec+
10. sequence_type          ::=  sequence t_name { slash | seq_type }
11. seq_type               ::=  a_type slash sequence
12. union_type             ::=  union t_name switch_by case_list slash union
13. switch_by              ::=  switch is quote switch_type quote
14. switch_type            ::=  integer | long | boolean | enumerated_type
15. case_list              ::=  a_case+ [default_case]
16. a_case                 ::=  case_head element_spec case_tail
17. case_head              ::=  case_label+
18. case_label             ::=  on val is quote string quote
19. case_tail              ::=  case_end+
20. case_end               ::=  slash on
21. default_case           ::=  default element_spec slash default
22. constant_element_spec  ::=  element_spec element_const_value
23. element_spec           ::=  element a_name a_type
24. element_const_value    ::=  minvalue is constant_integer_literal maxvalue is constant_integer_literal
25. constant_integer_literal ::=  quote integer_literal quote
26. complex_type_identifier ::=  identifier
27. t_name                 ::=  name is quote complex_type_identifier quote
28. a_name                 ::=  name is quote string quote
29. a_type                 ::=  type is quote { simple_type | complex_type_identifier } quote
30. option                 ::=  o
31. quote                  ::=  '"'
32. slash                  ::=  '/'
33. is                     ::=  '='
34. any_value              ::=  any_literal | complex_value
35. complex_value          ::=  enum_value | struct_value | seq_value | union_value
36. sequence_value         ::=  sequence {slash | seq_items slash sequence }
37. seq_items              ::=  seq_item+
38. seq_item               ::=  item index is quote integer_literal quote any_value slash item
```

Figure 3. Formal grammar definition of SLPL data types

its parameters have to be set. According to 3GPP specifications the parameters of the method are given as shown in *figure 5*.

The first parameter 'appl' of IpAppUserLocationRef type is the reference indication of the service logic instance subscribed to IpUserLocation interface. The requested reporting interval is presented by 'repInterval' that is of TpDuration. The 'users' parameter is of TpAddressSet type, that is a numbered set of users. The abstraction of the user is limited to his or her address presented by a structure of TpAddress type. The specific infor–mation requested by service logic concerning location of the users is contained in the 'request' parameter of TpLocationRe–quest type. The TpLocationRequest type is a structure of the following elements: requested accuracy ('requestedAccuracy'); response time ('requestedResponseTime'); requested altitude ('requestedAltitude'); type of location ('type') that might be current, current or last, or initial; and requested location method ('requestedLocationMethod').

The description of the other data types can be found in [9].

Figure 6 illustrates the SLPL definition of the arguments of the method periodicLocationReportingStartReq following the Mobility OSA API.

In SLPL predefined types 'Date', 'Time', 'DateTime' and 'Duration' based on the type float are introduced to handle with

```
<union name="TpAoCOrder"
    switch="TpCallAoCOrderCategory">
  <on val="P_CHARGE_ADVICE_INFO">
      <element name="ChargeAdviceInfo"
          type=" TpChargeAdviceInfo"/>
  </on>
  <on val="P_CHARGE_PER_TIME">
      <element name="ChargePerTime"
          type="TpChargePerTime"/>
  </on>
  <on val="P_CHARGE_NETWORK">
      <element name="NetworkCharge"
          type="string"/>
  </on>
</union>
```

Figure 4. SLPL description of the TpAoCOrder type

```
periodicLocationReportingStartReq(appl: in IpAppUser-
    LocationRef, users: in TpAddressSet, request : in
    TpLocationRequest, repInterval: in TpDuration) :
    TpAssignmentID
```

Figure 5. IDL definition of periodicLocationReportingStartReq method

a notion of 'real' time. Two operations 'CurrentDate' and 'CurrentTime' are used to acquire the value of the current time. Also, the concept of timers is adopted as a predefined type. When a timer is set using operations 'SetTime' or 'SetDate', a 'Time' or 'Date' value is associated with the timer respectively. When a timer is reset by 'Reset' operation, the associated value is lost. In many applications the decisions are made on the base of day of week so the enumerated type 'Day' is defined and is used to denote the days of week.

In OSA data types related to time and date are based on TpString. Time related data types in SLPL consider the fact that service logic often makes decisions on date, time and day of week. Operations on float type are more flexible than those on string type. The SLPL interpreter is responsible for type conversion.

Figure 7 shows time-related data types in SLPL.

IV. A SLPL Example of Service Logic Based on OSA APIs

To illustrate the suggested approach to service logic description we consider an application that exploits OSA interfaces for user interaction and call control. Imagine „Happy birth day" company that receives requests by clients and sends greetings by phone (e.g. musical or voice greetings on requested dates with the appropriate content for the occasion).

On receiving a request a record is created in a database. The service logic looks up daily in the database and retrieves the set of records corresponding to the current date. If there are some records the service logic creates a call to the particular number and plays the greeting.

The simplified service logic script in SLPL is shown in *figure 8*. Local types, variables and methods are defined in the definition part of the script. The database record type is defined as a structure of 3 elements: the date of greeting of type Date, the phone number to be dialed of type TpInt 32 and the greeting ID to be played of type TpString. A type of set of database records is defined as a numbered set of data elements. The database query result consists of number of records in the query result and the set of database records. Variables of the defined types are declared. A variable of type Duration represents the database look up interval and this interval is supervised by the use of variable of type Timer. The methods defined are local and include „DB_retrieve" and „DB_conversion", and also methods „routeRes" and „callEnded"supported by IpAppCall interface and method „sendInfoRes" of IpAppUICall interface.

The executive part of the service logic script is organized as an endless loop. First the current date is yielded. Then a SQL statement which retrieves all database records with date of greeting equal to the current date is created and the method „DB_retrieve" is invoked. The result returned is used as an argument of method „DB_conversion" invocation. The number of retrieved records and the set of these records are stored in the variable „a_DB_res" as a result of conversion. For each of the retrieved records the following is done: a call is created and routed to the phone number in the record and a user interaction is started that plays the greetings; after the end of the call, it is deassigned. When all of the retrieved records are browsed a timer is set with value equal to the database look up period.

```
<id name="users" type="TpAddressSet">
 <sequence>
  <item index="0">
   <structure>
    <element name="plan" value = "P_ADDRESS_PLAN_IP"/>
    <element name="addrstring" value = "164.23.7.3"/>
    <element name="name" value = ""/>
    <element name="presentation" value = "1"/>
    <element name="screening" value = "1"/>
    <element name="subaddrstring" value = ""/>
   </structure>
  </item>
  <item index="1">
   <structure>
    <element name="plan" value="P_ADDRESS_PLAN_E164"/>
    <element name="addrstring" value = "+359888010101"/>
    <element name="name" value = ""/>
    <element name="presentation" value = "2"/>
    <element name="screening" value = "2"/>
    <element name="subaddrstring" value = ""/>
   </structure>
  </item>
 </sequence>
</id>
<id name="request" type="TpLocationRequest">
 <structure>
  <element name="requestedaccuracy" value="2.0"/>
  <element name="requestedresponsetime">
   <structure>
    <element name="responsetime" value="0"/>
    <element name="timervalue" value="26951"/>
   </structure>
  </element>
  <element name="altituderequested" value="false"/>
  <element name="type" value="0"/>
  <element name="priority" value="0"/>
  <element name="requestedlocationmethod" value=""/>
 </structure>
</id>
<id name="reportinginterval" type="integer" value="250"/>
```

Figure 6. SLPL description of variables for starting periodic location reporting

```
<alias name="Date" type="Float"/>
<alias name="Time" type="Float"/>
<alias name="DateAndTime" type="Float"/>
<alias name="Duration" type="Float"/>
<type name="Day">
    <o>
      <element name="MO" minvalue="1" maxvalue="1"/>
      <element name="TU" minvalue="2" maxvalue="2"/>
      <element name="WE" minvalue="3" maxvalue="3"/>
      <element name="TH" minvalue="4" maxvalue="4"/>
      <element name="FR" minvalue="5" maxvalue="5"/>
      <element name="SA" minvalue="6" maxvalue="6"/>
      <element name="SU" minvalue="7" maxvalue="7"/>
    </o>
</type>
```

Figure 7. Time related data types in SLPL

```
<logic>
  <define>
    <types>
      <structure name="DB_record"> <!--a record in query result-->
        <element name="The_date" type="Date"/> <!--the date in query result-->
        <element name="Phone_num" type="TpInt32"/> <!-- phone num in query result-->
        <element name="greetingID" type="TpString"/> <!-- ID of greeting-->
      </structure >
      <sequence name="DB_records" item_type="DB_record"/>
      <structure name="DB_result"> <!-- DB query result-->
        <element name="SetOfRecords" type="DB_records"/> <!-- records in query result-->
        <element name="numRec" type="TpInt32"/> <!-- records number in query result-->
      </structure>
    </types>
    <variables>
      <id name="a_DB_res" type="DB_result"/> <!-- a database query result -->
      <id name="daily" type="Duration" val="1" /> <!-- DB look up period -->
      <id name="T_greetings" type="Timer" /> <!-- timer for DB look up period-->
      <id name="theSQLquery" type="TpString" /> <!-- SQL statement-->
      <id name="theSQLresult" type="TpString" /> <!-- SQL result-->
      <id name="DB_address" type="TpSrting"/> <!-- address of database server-->
      <id name="aDate" type="Date"/> <!--current date-->
    </variables>
    <methods>
      <!--  Database Query Handler 'IpAppLogic::DB_retrieve'
            Database Query Convertion Handler 'IpAppLogic::DB_conversion'
            Call Control 'IpAppCall::routeRes', User Interaction 'IpAppUICall::sendInfoRes'
            Generic Call Control 'IpAppCall::callEnded' -->
    </methods>
  </define>
  <execute>
    <!-- FRAMEWORK AUTHENTICATION -->
    <while test="true">
      <set refid="aDate"> <value> <CurrentDate/> </value> </set>
      <!-- 1.SET VALUE TO 'theSQLquery'  2.SET VALUE TO 'DB_address"-->
      <invoke>
        <method name="DB_retrieve">
          <arguments> <argument name="SQL_statement" valref="theSQLquery"/>
                      <argument name="DB_URL" valref="DB_address"/> </arguments>
          <returns> <set refid="theSQLresult"/> </returns>
        </method>
      </invoke><wait/>
      <invoke>
        <method name="DB_conversion">
          <arguments> <argument name="SQL_result" valref="theSQLresult"/></arguments>
          <returns> <set refid="a_DB_res"/> </returns></method>
      </invoke>
      <if> <condition test="numRec EQ 0" /> <!-- no retrieved records -->
        <then><goto label="skip_day"/> </then> </if>
      <while test="a_numRec GT 0"> <!-- while there are retrieved records -->
        <decrease refid="a_numRec " by="1" />
          <!-- 1. SET PARAMETERS  2. INVOKE Generic Call Control 'IpCall::createCall'
               3. INVOKE Call Control 'IpCall::routeReq'4. WAIT Call Control 'IpAppCall::routeRes'
            5. INVOKE UI 'IpUICall::sendInfoReq'
            6. WAIT UI 'IpAppUICall::sendInfoRes' 7. WAIT Call Control 'IpAppCall::callEnded'
            8. INVOKE Generic Call Control 'IpCall::deiassignCall' -->
      </while>
      <label name="skip_day"/>
      <invoke>
        <method name="SetDate">
          <arguments> <argument name="a_date" valref="aDate"/>
                      <argument name="a_dur" valref="daily"/>
                      <argument name="a_timer" valref="T_greetings"/></arguments>
          <returns/></method>
      </invoke>
      <wait/>
    </while>
  </execute>
</logic>
```

Figure 8. Simplified service logic script for greetings by phone

V. Conclusion

In this paper a new markup approach to service creation is presented. Following this approach the service logic can derive an added value from network functionalities accessible through APIs. The service logic is described by the use of an XML-based language SLPL that is platform independent, light-weight and suitable for 3rd party application development. As the

language is intended to be used with OSA APIs, services described in SLPL can benefit from network functions hiding network protocol complexity. On the contrary, to the other markup languages SLPL allows control over call-unrelated events such as change of position, user status, TCP/IP session, presence and availability and so on. The SLPL follows closely the architecture and APIs definitions developed by the OSA. Traditional value-added services provided in intelligent networks can be implemented by the SLPL and thus accessible also in IP-based networks. These language features get telecommunication service development near to the IT community and shorten time to market.

References

1. Bakker, J-L., D.Tweedie and M. Umnehopa. Evolving Service Creation; New Developments in Network Intelligence. Teletronikk, 2004.
2. Bakker, J., R. Jain. Next Generation Service Creation Using XML Scripting Languages. www.argreenhouse.com/papersjlbakker/bakker icc2002.pdf.
3. Falcarin, P., C.A. Licciardi. Analysis of NGN Service Creation Technologies. IEC Annual Review of Communications, 2003.
4. Lennox, J. and H. Schulzrinne. Call Processing Language Fra–mework and Requirement. 2000, http://www.ietf.org/rfc2824.txt.
5. Pencheva, E., I. Atanasov. XML-based Languages for intelligent Service Creation. ICEST'2005, Nish, Serbia and Monte Negro, Proceedings, 610-613.
6. Atanasov, I. Implementing Intelligent Network Services with the Call Processing Language.ICEST'2005, Nish, Serbia and Monte Negro, Proceedings, 606-609.
7. Atanasov, I., E. Pencheva. Service Creation Using a New Mark up Language. TELSIKS'2005, Nish, Serbia and Monte Negro, Pro–ceedings, 575-578. http://ieeexplore.ieee.org/xpl/free–abs_all.jsp.
8. Atanasov, I., E. Pencheva. A New Service Logic Processing Language. ELECTRONICS'2005, Sozopol, Bulgaria, Proceedings, book 1,150-155.
9. 3GPP TS 29.198-2. Open Service Access (OAS) Application Programming Interface (API); Part 2: Common Data Definitions.
10. 3GPP TS 29.198-5. Open Service Access (OAS) Application Programming Interface (API); Part 5: User Interaction Service Capability Feature (SCF).
11. 3GPP TS 29.198-6. Open Service Access (OAS) Application Programming Interface (API); Part 6: Mobility Service Capability Feature (SCF).

Manuscript received on 11.05.2006

Ivaylo Atanasov *received his M.S. degree in electronics from Technical University of Sofia, Bulgaria. He obtained his PhD degree in Telecommunication networks. His current position is Assistant Professor at Faculty of Telecommunications, Technical University of Sofia. His academic experience is in teaching courses on object-oriented programming, mobile networks and service creation technologies. The main research focus is development of open service platforms for next generation networks. He is an author of more than 30 scientific papers, text books and manuals.*

Contacts:
Faculty of Communications, Technical University of Sofia,
8 Kliment Ohridsky Blvd, 1756 Sofia , Bulgaria,
e-mail: iia@tu-sofia.bg

Evelina Pencheva *received her M.S. degree in mathematics from University of Sofia, Bulgaria. She obtained her PhD degree in Telecommunications from Technical University of Sofia. Since 1996 she is Associate Professor at Faculty of Telecommunications, Technical University of Sofia. Her academic experience is in teaching courses on telecommunication networks and service technologies. Her interests include next generation mobile applications and middleware platforms. She is a member of Specialized Science Council in Radio Electronics and Communication Technologies.*

Contacts:
Faculty of Communications, Technical University of Sofia,
8 Kliment Ohridsky Blvd, 1756 Sofia , Bulgaria,
e-mail: enp@tu-sofia.bg