SPL Language Specification

Joint work in the Phoenix Group (Labri, Bordeaux)

November 4, 2005

1 Lexical Conventions

Blanks

The following characters are considered as blanks: space, newline and horizontal tabulation. Blanks separate adjacent identifiers, literals and keywords that would be otherwise confused as a single identifier, literal or keyword. Apart from that, they are ignored.

Comments

Comments are C-like comments. They start with the characters `/*` and end with the characters `*/`. C++ comments style can also be used; all characters from the two characters `//` until the end of the line are considered as part of a comment. Comments are treated as blanks.

Special lexeme

There are several kinds of identifiers :

- *ident* which defines identifier (for variables, functions, branches and service name)
- *headerId* which defines SIP header in the RFC fashion, which means, with a colon at the end of the header name
- *eventIdent* which defines SIP event in the RFC fashion, which means, an alphanumeric sequence of characters with possibly some '.' inside when template event packages are used
- *uriScheme* which is defined as *scheme* in the RFC 3261

They use common definitions of *letter* and *digit*.

$$\begin{array}{lll} \textit{letter} & ::= & \texttt{A..Z} \mid \texttt{a..z} \\ \textit{digit} & ::= & \texttt{0..9} \end{array}$$

Identifiers

Identifiers are a sequence of letters, digits and _ (the underscore character) starting with a letter or an underscore. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of an identifier.

$$\textit{ident} \quad ::= \quad (\textit{letter} \mid \_)\ (\textit{letter} \mid \textit{digit} \mid \_)^*$$

Header names

Header names are a sequence of letters, digits and `-` (the minus character) starting with a letter and ending with a colon. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of a header name.

headerId ::= `#`*letter* (*letter* | *digit* | `_` | `!` | `%` | `*` | `-` | `+` | `'` | `` ` `` | `~`)* `:`

Event identifiers

Event identifiers are a sequence of letters, digits and special characters as defined in RFC 3265. A letter can be any of the 52 lowercase and uppercase letters from the ASCII set. The current implementation places no limit on the number of characters of an event identifier.

eventIdent	::=	*event-package* (`.` *event-template*)*
event-package	::=	*token-nodot*
event-template	::=	*token-nodot*
token-nodot	::=	(*letter* | *digit* | `_` | `!` | `%` | `*` | `-` | `+` | `'` | `` ` `` | `~`)+

URI Scheme

URI scheme are defined as in RFC 3261

uriScheme ::= *letter* (*letter* | *digit* | `-` | `+` | `.`)*

Integer Literals

An integer literal is a sequence of one or more digits. Integer literals are in decimal (radix 10).

integer ::= `0` | (`1..9` *digit**)

Prefix and Infix Operator

The following tokens are the SPL operators:

`=`	`+`	`-`	`*`	`/`	`&&`
`==`	`!=`	`<`	`>`	`>=`	`||`
`<=`	`!`	`match`	`notmatch`		

Keywords

The identifiers below are reserved keywords:

```
BODY          bool          branch        break         bye
cancel        case          continue      default       deploy
dialog        else          event         false         FIFO
foreach       forward       http          https         if
in            incoming      int           LIFO          local
mailto        match         notmatch      outgoing      parallel
pop           processing    push          reason        registration
remote        request       requestURI    response      return
select        service       sip           sips          string
time          true          type          undeploy      uninvite
unregister    unsubscribe   uri           void          when
with
```

The following character sequences are also reserved.

```
{  }  [  ]  (  )  <  >  |
=  :  ,  ;  .
```

Besides the above mentioned keywords, all *sipHeader*, *responseKind* and *SIPmethod* are also keywords and are written in uppercase.

Ambiguities

Lexical ambiguities are resolved according to the "longest match" rule: when a character sequence can be decomposed into tokens in several different ways, the resulting decomposition is the one with the longest first token.

2 SPL program

A SPL program consists of a service name and a *service* construct.

program	::=	`service` *ident* {*service*}
service	::=	`processing` {*declarations*$^?$ *session**}
session	::=	`registration` {*declarations*$^?$ *session**}
	|	`dialog` {*declarations*$^?$ *method*$^+$}
	|	`event` *eventIdent* {*declarations*$^?$ *method*$^+$}
	|	*method*
method	::=	*typExp* *direction*$^?$ *methodName*`(`*arg*$^?$`)` {*stmt*$^+$}
	|	*typExp* *direction*$^?$ *methodName*`(`*arg*$^?$`)` {*branch*$^+$}
branch	::=	`branch` *ident* (`|` *ident*)* {*stmt*$^+$}
	|	`branch default` {*stmt*$^+$}
direction	::=	`incoming`
	|	`outgoing`
methodName	::=	*SIPmethod*
	|	*ctrlMethod*
SIPmethod	::=	`ACK`
	|	`BYE`
	|	`CANCEL`
	|	`INVITE`
	|	`NOTIFY`
	|	`OPTIONS`
	|	`REACK`
	|	`REGISTER`
	|	`REINVITE`
	|	`REREGISTER`
	|	`RESUBSCRIBE`
	|	`SUBSCRIBE`
ctrlMethod	::=	`deploy`
	|	`undeploy`
	|	`uninvite`
	|	`unregister`
	|	`unsubscribe`

Notes :

- Without explicit *direction*, a method applies for both direction, incoming and outgoing.
- There is no SIP message associated with control methods, so headers are not accessible. These methods do not have a response return type but a `void` one.
- `unregister` is called after a `REGISTER` with a zero `Expire` field or after related timeout.
- `unsubscribe` is called after a `SUBSCRIBE` with a zero `Expire` field, a `NOTIFY` with `Subscription-State: terminated` header or after related timeout.
- `deploy` and `undeploy` can only appear directly in the processing block.

- Processing block can have only one registration block.
- Registration block can have only one session block.
- Registration block can have multiple event block, but each one must have an unique event identifier.

3 Type Expressions

typExp	::=	*simpleType*
	|	*modifier*$^{?}$ *simpleType*`<`*integer*$^{?}$`>`
	|	*ident*
simpleType	::=	`void`
	|	`bool`
	|	`int`
	|	`request`
	|	`response`
	|	`string`
	|	`time`
	|	`uri`
modifier	::=	`LIFO`
	|	`FIFO`

Notes :

- The sequence size must be fixed at compile time. A policy could be used to limit the size value.
- According to static properties, some information about a sequence (*e.g.* maximum size, starting position, current position) could be forgotten.
- The sequence head is on the left side, this is the first element to pop. According to the sequence kind, the push operation add an element on either the head or the tail of the sequence.
- If the modifier is not used, a sequence is a `LIFO` sequence.

4 Function call

functionCall	::=	*ident*`(`*expList*$^{?}$`)`
expList	::=	*exp* `(,` *exp*$)^{*}$

5 Know URI Kind

uriKind	::=	`https`
	|	`http`
	|	`mailto`
	|	`sips`
	|	`sip`
	|	`uri`
	|	*uriScheme*

6 Declarations

declarations	::=	*declarations declaration*
	|	*declaration*
declaration	::=	*typExp ident* (`=` *exp*)$^{?}$`;`
	|	*location typExp ident*`(`*args*$^{?}$`);`
	|	*typExp ident*`(`*args*$^{?}$`)` {*stmt*$^{+}$}
	|	`type` *ident* { (*typExp ident*`;`)$^{+}$}`;`
location	::=	`remote`
	|	`local`
args	::=	*arg* (`,` *arg*)*
arg	::=	*typExp ident*

Notes :

- Initialize an identifier with a `forward` expression is only allowed in a method, this is a signaling action. So, it cannot happen in blocks like `processing`, `registration` and `session` outside of a method.
- A function can only call another one which is already declared.

7 Statements

compound	::=	*stmt*
	|	{*stmt*$^+$}
stmt	::=	*place* `=` *exp*`;`
	|	*declaration*
	|	`return` *exp*$^?$ (`branch` *ident*)$^?$`;`
	|	`if(`*exp*`)` *compound*
	|	`if(`*exp*`)` *compound* `else` *compound*
	|	`when` *exp* `(`*when-headers*`)` *compound*
	|	`when` *exp* `(`*when-headers*`)` *compound* `else` *compound*
	|	`foreach(`*ident* `in` *exp*`)` *compound*
	|	`select(`*exp*`)` { *selectCase** *selectDefault*$^?$ }
	|	*functionCall*`;`
	|	`continue;`
	|	`break;`
	|	`push` *place* *exp*`;`
when-headers	::=	*when-header* (`,` *when-header*)*
when-header	::=	*headerId* *typExp* *ident* (`<-` *const*)$^?$
selectCase	::=	`case` *orList*`:` (*stmt*)*
selectDefault	::=	`default:` (*stmt*)*
orList	::=	*const* (`|` *const*)*
place	::=	*ident*(`.`*ident*)*
	|	*sipHeader*

Notes :

- `continue` and `break` are only allowed inside a `foreach`.
- Function `application` is the only expression that can also be a statement.
- `match` and `notmatch` are not allowed in a `case`
- The non-terminal `place`, is a place you can assign into, like a variable or a header field in a request.
- Header fields, in either a response or a request, are modified by the `with` expression.
- The expression in `when` can only be an identifier.

8 Expressions

```
exp       ::= ident(.ident)*
            | constant
            | sipHeader
            | unop exp
            | exp binop exp
            | (parallel)? forward (exp)?
            | exp with {messageField (, messageField)*}
            | (exp)
            | reason
            | BODY
            | requestURI
            | pop place
            | functionCall
unop      ::= !
            | -
binop     ::= +
            | -
            | *
            | /
            | <
            | >
            | ==
            | !=
            | <=
            | >=
            | &&
            | ||
            | match
            | notmatch

constant      ::= true
                | false
                | integer
                | "string"
                | uri
                | sequence
                | response
sequence      ::= <constant (, constant)*>
                | <>
uri           ::= 'uriKind:uriString'
messageField  ::= reason=exp
                | headerId exp
```

Notes :

- The expression, used in `with` expression, must be of response or request kind.

- The expression, used in `pop` expression, must be of sequence kind.
- `match` and `notmatch` use POSIX regular expressions.

9 SIP headers

If SIP headers, specified in the RFC 3261 and RFC 3265, are mandatory in at least one method, they are integrated in the SPL compiler as keywords. SIP headers are restricted in terms of type or access rights (NA, RO, RW). Header keywords refer to request headers only. When a header is not mandatory in a request and for all responses, the `when` statement allow a protected access to read headers in messages. Headers can only be modified with the `with` expression.

sipHeader ::= `CALL_ID`
| `CONTACT`
| `CSEQ`
| `EVENT`
| `FROM`
| `MAX_FORWARDS`
| `SUBSCRIPTION_STATE`
| `TO`
| `VIA`

10 Constant Responses

Standard responses in RFC 3261 and RFC 3265 are defined in the language.

response	::=	/ERROR*responseError*$^{?}$
	|	/SUCCESS*responseSuccess*$^{?}$
responseSuccess	::=	/OK
	|	/ACCEPTED
responseError	::=	/CLIENT*clientErr*$^{?}$
	|	/GLOBAL*globalErr*$^{?}$
	|	/REDIRECTION*redirectionErr*$^{?}$
	|	/SERVER*serverErr*$^{?}$
redirectionErr	::=	/ALTERNATIVE_SERVICE
	|	/MOVED_PERMANENTLY
	|	/MOVED_TEMPORARILY
	|	/MULTIPLE_CHOICES
	|	/USE_PROXY

clientErr ::= /ADDRESS_INCOMPLETE
| /AMBIGUOUS
| /BAD_EXTENSION
| /BAD_REQUEST
| /BUSY_HERE
| /CALL_OR_TRANSACTION_DOES_NOT_EXIST
| /EXTENSION_REQUIRED
| /FORBIDDEN
| /GONE
| /INTERVAL_TOO_BRIEF
| /LOOP_DETECTED
| /METHOD_NOT_ALLOWED
| /NOT_ACCEPTABLE_HERE
| /NOT_ACCEPTABLE
| /NOT_FOUND
| /PAYMENT_REQUIRED
| /PROXY_AUTHENTICATION_REQUIRED
| /REQUESTURI_TOO_LONG
| /REQUEST_ENTITY_TOO_LARGE
| /REQUEST_PENDING
| /REQUEST_TERMINATED
| /REQUEST_TIMEOUT
| /TEMPORARILY_UNAVAILABLE
| /TOO_MANY_HOPS
| /UNAUTHORIZED
| /UNDECIPHERABLE
| /UNSUPPORTED_MEDIA_TYPE
| /UNSUPPORTED_URI_SCHEME

serverErr ::= /BAD_GATEWAY
| /MESSAGE_TOO_LARGE
| /NOT_IMPLEMENTED
| /SERVER_INTERNAL_ERROR
| /SERVER_TIMEOUT
| /SERVICE_UNAVAILABLE
| /VERSION_NOT_SUPPORTED

globalErr ::= /BUSY_EVERYWHERE
| /DECLINE
| /DOES_NOT_EXIST_ANYWHERE
| /NOT_ACCEPTABLE

11 Examples

Some examples are given to illustrate the possible constructions.

11.1 A simple service

The simplest example is to forward all calls (incoming and outgoing). It illustrates how a SPL program is formed. We must have at least one method or an included session in a session section.

```
service simple {
  processing {
    registration {
      dialog {
        response incoming INVITE() {
          return forward;
        }

        response outgoing INVITE() {
          return forward;
        }
      }
    }
  }
}
```

11.2 Declarations

An other simple example is to forward to a new SIP URI. This URI can be declared at several place.

```
service declare1 {
  processing {
    uri us = 'sip:phoenix@barbade.enseirb.fr';

    registration {
      response incoming INVITE() {
        return forward us;
      }
    }
  }
}
```

```
service declare2 {
  processing {
    registration {
      uri us = 'sip:phoenix@barbade.enseirb.fr';

```

```
      dialog {
        response incoming INVITE() {
          return forward us;
        }
      }
    }
  }
```

```
service declare3 {
  processing {
    dialog {
      uri us = 'sip:phoenix@barbade.enseirb.fr';

      response incoming INVITE() {
        return forward us;
      }
    }
  }
}
```

```
service declare4 {
  processing {
    dialog {
      response incoming INVITE() {
        uri us = 'sip:phoenix@barbade.enseirb.fr';

        return forward us;
      }
    }
  }
}
```

11.3 Sequences

```
service seq1 {
  processing {
    dialog {
      response incoming INVITE() {
        uri<> us = <'sip:student.1@enseirb.fr',
        'sip:student.2@enseirb.fr',
        'sip:student.3@enseirb.fr'>;

        return forward us;
      }
    }
  }
}
```

```
service seq2 {
  processing {
    dialog {
      response incoming INVITE() {
        uri<3> us = <'sip:student.1@enseirb.fr',
        'sip:student.2@enseirb.fr',
        'sip:student.3@enseirb.fr'>;

        return parallel forward us;
      }
    }
  }
}
```

```
service seq3 {
  processing {
    dialog {
      response incoming INVITE() {
        LIFO uri<3> us = <'sip:student.1@enseirb.fr',
        'sip:student.2@enseirb.fr',
        'sip:student.3@enseirb.fr'>;

        return forward us;
      }
    }
  }
}
```

11.4 Matching

```
service matching {
  processing {
    dialog {
      response incoming INVITE() {
        if (requestURI match ".*@phoenix\.labri\.fr") {
          return forward 'sip:phoenix-contact@enseirb.fr';
        }
        else {
          return forward;
        }
      }
    }
  }
}
```

11.5 Use of header field

```
service header_field {
```

```
    processing {
      dialog {
        response incoming INVITE() {

          if (FROM == 'sip:wife@fwd.com') {
            return forward;
          }
          else {
            return forward 'sip:phoenix-contact@enseirb.fr';
          }
        }
      }
    }
}
```

11.6 Make a response

```
service moved {
  processing {
    dialog {
      response incoming INVITE() {
        return /ERROR/REDIRECTION/MOVED_PERMANENTLY
        with {reason = "I'm now working at ENSEIRB",
          #contact: 'sip:me@enseirb.fr' };
      }
    }
  }
}
```

11.7 Safe access to headers

```
service boss_secretary {
  processing {
    dialog {
      response outgoing INVITE(request rq) {
        if (TO == 'sip:my.boss@big-corp.com') {
          response x = forward;
          select (x) {
          case /ERROR:
            when rq (#subject: string sub) {
              rq = rq with {#subject: "The boss is unavailable ! - "+ sub};
            }
            when x (#contact: uri r_contact) {
              if (r_contact != CONTACT) {
                return forward r_contact;
              }
            }
```

```
              return forward 'sip:boss.secretary@big-corp.com';

            case /SUCCESS: return x;
            }
          }
          else {
            return forward;
  } } } } }
```

11.8 Walk through a sequence

```
1 service foreach_seq {
    processing {
      dialog {
        response incoming INVITE() {
5
          uri<> students =
          <'sip:student.1@enseirb.fr',
          'sip:student.2@enseirb.fr',
          'sip:student.3@enseirb.fr'>;
10
          if ( FROM == 'sip:wife@fwd.com' ) {
            return forward;
          }
          else {
15          foreach (student in students) {
              if ( FROM == student ) {
                return /ERROR/REDIRECTION/MOVED_TEMPORARILY
                with {reason = "Busy Here contact Bob",
                  #contact: 'sip:bob@enseirb.fr'};
20            }
            }
            return forward 'sip:phoenix-contact@enseirb.fr';
          }
        }
25    }
    }
  }
```

11.9 Selection according to an expression

```
1 service selection {
    processing {
      dialog {
        response incoming INVITE() {
5         select (FROM) {
            case 'sip:9336@fwd.com' :
```

```
            return /ERROR/CLIENT/BUSY_HERE
            with {reason = "I do not want to call you"};
            default :
            return forward;
          }
        }
      }
    }
}
```

11.10 Sequence manipulations

11.10.1 use of push

```
service push_seq {
  processing {
    dialog {
      response incoming INVITE() {
        uri<> students =
        <'sip:student.1@labri.fr',
        'sip:student.2@labri.fr',
        'sip:student.3@labri.fr',
        'sip:student.1@inria.fr',
        'sip:student.2@inria.fr',
        'sip:student.3@inria.fr'>;

        uri<100> st_inria;
        uri<100> st_labri;

        if (requestURI == 'sip:students@labri.fr') {
          foreach (student in students) {
            if (student match ".*@labri\.fr") {
              push st_labri student;
            }
          }
          return forward st_labri;
        }
        else if (requestURI == 'sip:students@inria.fr') {
          foreach (student in students) {
            if (student match ".*@inria\.fr") {
              push st_inria student;
            }
          }
          return forward st_inria;
        }
        return forward;
      }
```

```
        }
35    }
    }
```

11.10.2 use of pop

```
1   service pop_seq {
      processing {
        dialog {
          response incoming INVITE() {
5           uri<> students =
            <'sip:student1@labri.fr',
            'sip:student2@labri.fr',
            'sip:student3@labri.fr'>;

10          uri<2> callee;

            foreach (student in students) {
              push callee pop students;
            }
15          return forward callee;
          }
        }
      }
    }
```

11.11 Escape when walking through a sequence

Two methods are available to do this :

- With a counter and break statement (*e.g.* "`i`" and "`break;`")
- With a fix size sequence (*e.g.* "`uri<2> st_labri;`")

```
1   service foreach_break {
      processing {
        dialog {
          response incoming INVITE() {
5           uri<> students =
            <'sip:student.1@labri.fr',
            'sip:student.2@labri.fr',
            'sip:student.3@labri.fr',
            'sip:student.1@inria.fr',
10          'sip:student.2@inria.fr',
            'sip:student.3@inria.fr'>;

            uri<100> st_inria;
            uri<2> st_labri;
```

```
15          int i = 0;

            if (requestURI == 'sip:students@labri.fr') {
              foreach (student in students) {
                if (student match ".*@labri\.fr") {
20                push st_labri student;
                }
              }
              return forward st_labri;
            }
25          else if (requestURI == 'sip:students@inria.fr') {
              foreach (student in students) {
                if (student match ".*@inria\.fr") {
                  push st_inria student;
                  i = i + 1;
30              }
                if (i==2)
                  break;
              }
              return forward st_inria;
35          }
            return forward;
          }
        }
      }
40  }
```

11.12 Branching

```
1   service branching {
      processing {
        dialog {
          response incoming INVITE(request rq) {
5
            if (FROM == 'sip:my.wife@home.net') {
              rq = rq with {#priority: "urgent"};
              return forward branch wife;
            }
10          else {
              rq = rq with {#priority: "urgent"};
              return forward;
            }
          }
15
          response BYE(request rq) {
            branch wife {
              rq = rq with {#priority: "urgent"};
```

```
            return forward;
          }
          branch default {
            rq = rq with {#priority: "non-urgent"};
            return forward;
  } } } } }
```

11.13 External functions

When an external function is declared, its location must be declared too. It can be either `remote` or `local` according to the fact the function is executed on a remote machine or on the same one as SPL program.

```
service anti_spam {
  processing {

    remote bool isSpam(uri from);

    dialog {
      response incoming INVITE() {
        if(isSpam(FROM)) {
          return /ERROR/GLOBAL/DECLINE with {
                    reason ="Your call was considered as a spam call, and was blocked."};
        }
        else {
          return forward;
  } } } } }
```

11.14 Registration

```
service log_presence {
  processing {
    int db;

    local int open_DB(string);
    local int log_in_DB(int, string, uri);
    local void close_DB(int);

    void deploy() {
      db = open_DB("sql.enseirb.fr");
    }

    registration {
      uri who;

      response REGISTER() {
        who = FROM;
```

```
            log_in_DB(db, "sign in", who);
            return forward;
          }

          void unregister() {

            log_in_DB(db, "sign out", who);
          }
        }

        void undeploy() {
          close_DB(db);
    } } }
```

11.15 Special methods

```
1   service limit_fwd {
      processing {
        dialog {
          response INVITE(request rq) {
            rq = rq with {#max-forwards: 2};
            return forward;
          }

          response REINVITE(request rq) {
            rq = rq with {#max-forwards: 2};
            return forward;
    } } } }
```

11.16 Event

```
1   service event_srv {
      processing {

        registration {

          event presence {
            response incoming SUBSCRIBE() {
              if (FROM == 'sip:my-boss@big-corp.com') {
                return /ERROR/SERVER/SERVICE_UNAVAILABLE;
              }
              else {
                if (FROM == 'sip:my-wife@home.net') {
                  return forward branch wife;
                }
                else {
                  return forward;
```

```
            }
          }
        }

        response outgoing NOTIFY(request rq) {
          branch wife {
            rq = rq with {#priority: "urgent"};
            return forward;
          }
          branch default {
            return forward;
} } } } } }
```