SLIC
Standard Language for Implementation Conventions

Alex Th. de Jong
dejong@snad.ncsl.nist.gov

October 27, 1994

1 Introduction

This paper discusses the first version of a Standard Implementation Language for Implementation Conventions. The language can describe a standard ([3]/ [4]) as well as an Implementation Convention in a flexible and efficient way.
This paper will give an overview of the language's capabilities and is followed by part of the syntax definition. The syntax definition part gives some some examples to clarify the usage.

- Note:
  1. The purpose of this paper is to present SLIC's functionality in order to get comments; not to give a full syntax definition.
  2. In order to place this document in its proper context it might be necessary to read the project description[1] first.

2 Language overview

A full Implementation Convention (IC) normally consists of two (or more) files: the standard from which it is derived and the adjustments that describe the IC.

The former consists of a straightforward grammar that is effective as well as efficient. It is effective because it describes the standard both in a computer- and in a human-readable format; it is efficient because overhead is kept to a minimum. The object information contained corresponds to codes, elements, composites, segments and transaction sets as defined in the standard. The objects can also carry syntactic and semantic notes as defined in the standard[3].

The latter file, which contains the delta between the IC and the standard, is based on derived objects [5]. Within derived objects, an IC designer redefines objects and adds constraints. Besides re-defining objects, the designer can also create new definitions of objects and use them throughout the IC. This enables an IC to be fully customized for specific use. Object constraints are expressed in a predefined grammar. This grammar specifies syntactic as well as semantic constraints.

In figure 1 the model shows how the two definitions, standard and IC, are related to each other. The IC shell modifies and adds objects to the standard before passing the information to the user. The combined definitions provide all the information needed to describe the IC.

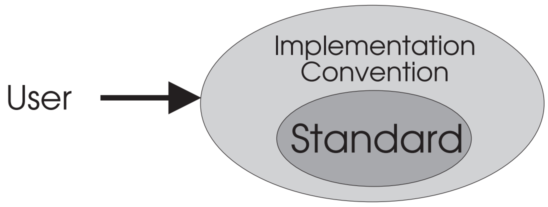

Figure 1: *SLIC overview*

The language can be used to distribute standards and ICs in computer readable file format. Commercial EDI software can use this information to upgrade their translators and applications for new releases of the standard. It will also allow EDI software to adjust to specific ICs in a fully automated manner (i.e. computer-computer without human intervention). A concept presented in [6] could very well be implemented by use of SLIC. SLIC's customized transaction set description capability is similar to the ICSDEF (Interchange Structure Definition) message. The "intelligent" translator could use a central registry to obtain an IC instead of sending it along (by means of an ICSDEF) with the original interchange.

In order to do the above effectively, a front-end is needed that interfaces proprietary software with the language. However, most commercially available systems already use some kind of proprietary intermediate file format. Upgrading intermediate formats to SLIC is a relatively small step which will partly solve interoperability problems in current EDI implementations.

Parts of the front-end SLIC interface will be available in public domain. A lexical analyzer and parser for SLIC are in development at NIST. These tools provide a generic Application Programmer's Interface (API). The API can be used for effective access of information. Only a small front-end link has to be created to interface proprietary software with the API.

The Transaction Set Development Tool (TSDS) [2], now using Script, will be upgraded in May 1995 to output SLIC as intermediate file format. The TSDS includes editors capable of creating fully customized transactions sets as well as defining the standard transaction sets.

3 Language capabilities

The language capabilities are divided into object related, constraints and miscellaneous. Each section lists its capabilities and gives a short example or reason why SLIC supports the capability. In section 4, examples are given of how to use and synactically implement the listed constructs.

Capabilities available for derived objects:

- *Multiple derivations of the same object.* Multiple objects can be derived from the same standard object. The derived objects will all have the same "base" object attributes but they are used differently within the IC.
- *Derivation from derived objects.* The designer can derive new objects from objects defined in the IC as well as from the underlying standard (if there is one). This allows a designer to build a new IC taking an old IC as starting point (instead of starting from scratch or a standard). Tools can automatically map successive ICs into one. This construct will be useful when EDI software does not follow up on each new release of an IC but only upgrades for successive ICs released in some time period.
- *Overwriting of standard attributes.* The name of an element can be overwritten to fit better in a specific context (i.e. segment). A user can understand the segment (and transaction set) more easily when names (or other attributes) are defined for a specific context instead of their dictionary names.
- *Addition of new items.* An extra code value is allowed within an element. Within the derived code the additional value can be added to the standard list. The element will use this extended set instead of the original set.
- *Suppression of items locally.* Only a limited number of elements within a segment is used within the IC. Within the derived segment object, unused elements can be marked as not used.

Semantic constraints place restrictions on the data element values; syntactic constraints force elements (or segments) to be used or omitted. The constraint[1] grammar defines the following constructs:

- *Assignment* $(=)$
- *Comparison operators* $(==, !=, <, >, \geq, \leq)$
- *Simple arithmetic operators* $(+, -, /, *)$
- *Logical operators* (&&(and), |(or), !(not))
- *Function calls*
- *Iteration operators* (while- and for-loops)
- *Conditional operators* (if-then-else constructs and syntax notes[3])

The operators allow for a wide range of constraint specification. The operators and functions can be used to set values of elements without user intervention. It depends upon the designer as to what to specify within the constraint.

Miscellaneous functionality:

[1]Constraints can be used at every level within the language (i.e. they can be defined within every object).

- *Defining proprietary software attributes.* The language includes constructs to define proprietary software variables. These constructs are provided to define values related to specific applications while remaining compatible with others. An example of usage is to define graphical information inside objects. This information tells a graphical interface how to display an object.

- *Suppressing items globally.* Suppressing an item from a list can be done at a local (i.e. inside an object) as well as a global level. This suppresses the object from the standard dictionary and therefore from all other objects that are using it. This can be useful to suppress an NTE segment for all transaction sets, instead of suppressing all NTE segments locally. One statement will suppress the segment in all transaction sets.

4 Syntax

4.1 Standard

The syntax to define a Standard in SLIC is simple and based on objects. Objects have a unique identifier and several attributes. Attributes will be assigned within the definition section of an object. The syntax in BNF form is the following:

object := objectId ':' identifier { standard }[2] definition ';'

identifier := string { '[' integer ']' } [3]
standard := '(' string ',' integer ',' integer ')'
definition := '{' { attribute } '}'
attribute := attributeId '=' (value | list) ';'

objectId := 'code' | 'element' | 'composite' | 'segment' | 'transaction'
attributeId:= 'name' | 'purpose' | 'delist' | string[4] | ...

The following example shows more clearly how to use the grammar. It defines element 143 for standard X12, version 3, release 30. The attributes (e.g. name, purpose, etc.) define all the information for element 143.

```
element : 143 (X12, 3, 30)
{
  name="Transaction Set Identifier Code";
  purpose="Code uniquely identifying a transaction set";

  type=identifier(3, 3);      -- type definition --
  codeId=143;                 -- reference to codes --
};
```

The structure to define other objects is similar to an element definition. Defining transaction sets (including loops) is a little more complex because there are some additional sequence numbers required. The IC uses these numbers to refer to a particular segment in a standard.

[2]Only used when defining an object from the standard.
[3]The repetition of the integer shows the depth of derivation. Only used within ICs.
[4]Proprietary software attribute.

4.2 Convention

The IC syntax uses the same syntax as the standard (Note: a few different options are used). At the top of the IC, a list indicates which standard and other ICs the current IC is derived from. This part also identifies the name, version and release of the IC to be defined. After this IC identification block, the designer can derive from objects as defined in the standard or add new customized objects. The following examples show how to implement the capabilities listed in section 4 at code and element level.

```
convention : NIST(0, 0)      -- NIST IC, version 0, release 0 --
{
  X12(3, 30);                -- Base standard is X12, version 3, release 30 --
};

code : 143[1]                -- Code derived from standard code 143 --
{
  valuelist=
  { "999" },                 -- Suppress code value 999 --
  { "new", "New code value" };        -- Add new value --
};
```

Adding new items and suppressing items is done for code *143*. Code value *999* is not available within code definition *143[1]* and code value *new* can be used instead (including all other code values defined in standard code *143*).

```
element : 143[1]             -- First derived element --
 {                            -- Overwrite original name --
   name="Transaction Set Identifier Code with additional values";
   codeId=143[1];             -- Use first derived code list --

   constraintlist=            -- Examples of constraints --
   { 1, "(value=\"850\")" }, -- Default value is 850 --
   { 2, "((value==\"850\") || (value==\"851\") || (value==\"852\"))" };
                              -- Only 850, 851 and 852 are valid values --
 };

 element : 143[2]             -- Second derived element --
 {                            -- Overwrite original name --
   name="Transaction Set Identifier Code for use within specific segment";
   style=BoxWithBorder;       -- Graphical environment atrribute --
 };
```

Multiple derivation is shown for element *143*. There are two slightly different versions of *143* available: *143[1]* and *143[2]*. New objects derived from these, could be created and would have the following names: *143[1][1]*, *143[1][2]*, etc. The example does not show this. Overwriting the element name is done for both derived *143* elements and a proprietary software attribute *style* is defined for element *143[2]* (*style*. Some examples of how to use constraints are placed in element *143[1]*.

```
element : 999                -- Defining a new element --
{
  name="New IC element";
  purpose="New IC element only for use within NIST IC";
  default=9.50;              -- Default value is 9.50 --
  type=numeric(0, 5, 2);     -- Minimum, maximum, decimal point position --
};
```

The last example shows how a new element is added to the IC. Element *999* is not available in the standard but can be used within the IC. Fully customized objects can be defined and used throughout the IC.

The constructs to define other objects (e.g. composites, segments or transaction sets) are similar to those given in the examples. Of course, different objects have different identifiers and attributes. A full syntax description and constraint grammar are in development.

5 Flexibility and limitations

Although SLIC is able to describe almost any change and addition to to the standard, proprietary software may not be able to handle all these changes and additions. A simple example of a limitation could be the following:

> Default data element values can be specified within the IC. However, the translator that uses the SLIC interface does not have space to store these default values. Default values can be specified but may not be used within the application.

The flexibility built into the language can be used in several ways. Limited usage would be, to use only the standard describtion capability. This will allow proprietary software to upgrade automatically for new releases of the standard. A more powerful use of the language is the constraint definition capability. Full use of this feature will enable applications to enforce extended syntactic as well as semantic constraints at the IC level.

6 Conclusion

SLIC describes most commonly used variations on a standard. Within the IC, an additional level of constraints can be specified that enforces syntactic and semantic rules. The generic SLIC tools allow users to access and use this information to upgrade their EDI applications automatically. The tools and the language are designed to support a large range of EDI translators and tools. However, not all applications will be able to use the full functionality of the language. This will depend on the specific application that is using the tools. The language and tools are flexible enough to allow for limited as well as poweful usage.

7 Abbreviations

API	Application Programmer's Interface
EDI	Electronic Data Interchange
SEF	Standard Exchange Format
SLIC	Standard Language for Implementation Conventions
NIST	National Institute of Standards and Technology
IC	Implementation Convention
IMPDEF	Implementation guideline Definition message
ICSDEF	Interchange Structure Definition

References

[1] A. Th. de Jong. *SLIC Standard Language for Implementation Conventions*. NIST/University of Twente Project description, June 1994.

[2] R. Sijelmassi, O. Kim, and B. Strausser. *EDI Development Tools Design Document*. NIST Technical Report CSL/SNA-92/2.

[3] Data Interchange Standards Association. *EDI X12 Standards - Draft Version 3, Release 3*. DISA-ASC X12S/92-707, Alexandria, VA, December 1992.

[4] United Nations. *EDI 1992-Trial-Directory*. Pan American EDIFACT board, TRADE/WP.4/R.841/Rev.1 (92.1), 1992.

[5] B. Stroustrup. *The C++ Programming Language*. Second Edition, ISBN 0-201-53992-6, 1991.

[6] K. Steel. *Matching functionality of interoperating applications: another approach to EDI standardisation*. University of Melbourne, ISO/IEC JTC1/SC30 IT11/7/94-103, July 1994.