SANKHYA Translation Framework™

- **User Guide**
- **Reference Manual**

Part No. 10030106-006

SANKHYA Translation Framework™

- **User Guide**
- **Reference Manual**

Sankhya Technologies Private Limited

Part No. 10030106-006

Table 1: Revision History

Revision number	Revision History	Date
001	STF 1.0 Release	10 Jan 2003
002	Updated for STF 1.0 Beta2	06 May 2003
003	Updated for STF 1.0 Beta3	16 Feb 2004
004	Upadted for STF 1.1 Release	17 Dec 2004
005	Updated for STF 1.1A release	07 Feb 2005
006	Updated for STF 1.1 A release	23 Feb 2005

Contents

Preface

SANKHYA Translation Framework (STF) is a completely novel framework for building model-driven translation/transformation tools and applications. It can be used to automate EAI activities like document and message processing, protocol conversion, SQL database to XML transformations, C++ and Java code generation, server page processing, data conversion and adapter development.

STF includes a powerful translation modeling language - the SANKHYA Translation Modeling Language (STML), and a set of tools (STML Line Translator, STML Server, C++ API) that automatically converts information in one format to any other format described using STML.

This document contains two parts:

Part-1 *User Guide* provides the user with the descriptions on how to invoke the STML Line translator (‘***st***’), STML Server (‘***stml_server***’) along with the options supported. It also describes the steps necessary for creating a model file using examples.

Part-2 *Reference Manual* provides the specification of the SANKHYA Translation Modeling Language and describes the various elements of STML.

Audience

This document provides the users of STF with the STML specifications and guidelines on creating the model file. It also explains the basic concepts and terms used in STML and how the various options of ‘*st*’ and ‘*stml_server*’ can be used for translation.

Notational Conventions

The guide follows the following conventions

%	-	The ‘percentage’ sign denotes a Unix environment.
>	-	The ‘greater than’ symbol represents a DOS/Windows environment.
Italic	-	The words given in italics represents an option.
code	-	The guide differentiates the normal text from a program code through this color. Any code, part of a code, input, output, command line statements in this document, will be represented using this color.
...	-	Indicates that some portion of the material has been removed to simplify the description.
[ ]	-	Indicates an optional argument that can be used in the command line.

USER GUIDE

Sankhya Technologies Private Limited

Part 1 - User Guide

1.1 Introduction

Translators are tools which convert information present in an input representation to equivalent information in an output representation using a set of rules supplied as input. Starting from language translations, a translation framework can be used in diverse application areas such as document conversion, script conversion, data exchange in Enterprise Application Integration (EAI).

SANKHYA Translation Framework (STF) is a completely novel framework for describing and performing model-driven translations and transformations. STF can be used for automating EAI activities like document and message processing, protocol conversion, data conversion, adapter development and natural language translation.

1.1.1 Overview

SANKHYA Translation Framework includes the following translation tools or C++ API for developing an application.

• st	-	STML Line translator
• stml_server	-	STML Server

STML Line Translator, '***st***' is a model driven translation and transformation tool that can both parse information (document, message, data) in one representation and automatically translate the information to any other representation based upon the specified Sankhya Translation Modeling Language (STML) model.

STMLServer, ‘***stml_server***’ is a CORBA based document server that can be used to transform or translate documents from one format to another based upon the specified STML model(s) in a distributed environment.

1.1.2 STF editions

STF is available as three editions,

- STF Command Line Edition ‘***st***’ (STML Line Translator)
- STF Client-Server Edition ‘***stml_server***’ (STML Server)
- STF Developer Edition (C++ API)

The command line edition of STF (st) and the Client Server edition of STF (stml_server) supports pluggable streams modules. Standard streams modules include file, string, symbol and database (ODBC) streams.

Additional streams for HTTP, FTP, IMAP can be quickly developed using the standard STML streams interface. Using the ODBC streams support, STF can be used to source data from a database and convert the same using an appropriate STML model to an XML or text document.

The STF developer edition provides C++ API for developing complex applications.

1.1.3 Hosts supported

The following hosts are supported by STF Command Line edition, STF Client Server edition and STF developer edition.

- Solaris 2.7
- Red Hat Linux 7.2
- Windows NT and Windows 2000

1.1.4 STF Features

The following are the features of SANKHYA Translation Framework.

- Model driven transcoding
- Automated data conversion and mapping
- Ready to use translation tools
- Pluggable streams module
- Powerful C++ API for complex applications
- Reverse translation
- Powerful Modeling Language (STML)

Unlike most existing XML technologies like XSLT, STML enables the transformation of information between different formats using a single model that describes all the formats.

1.2 SANKHYA Translation Modeling Language

SANKHYA Translation Modeling Language (STML) is a simple yet powerful language for modeling document formats, message formats and language structures.

STML allows multiple formats of data to be described in a single model description. This feature can be used to build model driven tools which can translate and transform from one format to any other format that has been described using STML.This finds application in Enterprise Application Integration, Program translation tools (like code generators, assemblers, binary translator) and natural language translation tools.

STML allows the specification of the hierarchies of structured information and the description of each node in the hierarchy in several different representations (different languages, different data formats etc). STML supports various data types for representing concepts of the input domain naturally. For detailed information on STML, please refer the “Reference Manual”.

1.3 STML Line Translator - ‘*st*’

STML Line Translator is a model driven translation and transformation tool that can both parse information (document, message, data) in one representation and automatically translate the information to any other representation.

For reversible transformations, ***’st’*** can recreate the original document, message or data from the transformed document. This is extremely useful when converted documents are edited and the original document needs to be updated with the changes.

The information to be translated is modeled using a modeling language called the SANKHYA Translation Modeling Language (STML). STML allows modeling of information in several different representations using a single model.

1.4 STML Client-Server Translator - ‘*stml_server*’

STMLServer is a CORBA based document server that can be used to transform or translate documents from one format to another based upon the specified Sankhya Translation Modeling Language (STML) model(s) in a distributed environment.

Any CORBA compliant client can access the STMLServer in a distributed environment. STMLServer also includes database (DB) stream support, model stream support, logical data base (LDB) support, input file stream support and directory stream support.

1.5 Setting STF environment

1.5.1 Using STML Line Translator

1.5.1.1 Setting up STF host development environment

To set the STF host development environment, the following environment variables should be set.

• STF_HOME	- Directory path pointing to the root of installation
• LD_LIBRARY_PATH	- Directory path pointing to the location of dynamic library.

For Solaris hosts:

STF_HOME is <INSTALL_DIR>/sankhya/stf
LD_LIBRARY_PATH is /usr/local/lib:$STF_HOME/lib/sol2

For Linux hosts:

STF_HOME is <INSTALL_DIR>/sankhya/stf
LD_LIBRARY_PATH is /usr/local/lib:$STF_HOME/lib/linux

For Windows hosts:

STF_HOME is <INSTALL_DIR>

where,
<INSTALL_DIR> is the directory where STF tools are installed.

By default STF is installed under /opt/ in Unix and C:\sankhya\stf in Windows NT/2000.

In order to set the variable STF_HOME and PATH variable on Unix, source the Unix shell script *stf.csh* in the STF tools installation root directory.

```
% source <STF_HOME>/stf.csh
```

On Windows host, run *stf.cm*d, in the STF tools installation directory. In a Windows Command Prompt, type

```
> %STF_HOME%\stf.cmd
```

1.5.2 Using STML Client-Server Translator

1.5.2.1 Setting up STF host development environment

To set the STF host development environment, please refer section 1.5.1.1.

1.5.2.2 Setting up Varadhi development environment

STMLServer is a CORBA based document server that can be used to transform documents from one representation to another based on the model file. To invoke the stml_server, environment for SANKHYA Varadhi, an Object Request Broker should be set.

To work with Varadhi, the following environment variable should be set.

VARADHI	- Points to root of installation
VARADHI_HOST	- Host platform name like sol2, linux, win32.

In order to set these variables and PATH variable on Unix, source the Unix shell script *varadhi.csh* in the SANKHYA Varadhi installation root directory.

```
% source <varadhi_root>/varadhi.csh
```

On Windows host, run varadhi.cmd instead. In a Windows Command Prompt, type

```
> <varadhi_root>\varadhi.cmd
```

where,

<varadhi_root> is the directory where SANKHYA Varadhi is installed.

For detailed information on SANKHYA Varadhi, please refer the “SANKHYA Varadhi User guide and Reference manual”

1.6 Usage Examples

Here is the complete overview of the steps to be followed to perform translation using STF.

Step-1: Create a model file.
Step-2: Create the input file to be processed.
Step-3: Invoke the translator with the model file and input file.

The below section explains each step listed above in detail for the STML Line Translator and the STML Client-Server Translator.

1.6.1 Sample using STML Line Translator - ‘array’

STML Line Translator samples can be found under the directory $(STF_HOME)/ samples/stml. The ‘**array**’ sample explained below, uses the array.md file as the model description file and array.in as the input file.

Step-1: Create a model file

Contents of array.md file:

```
STMLTextTokens CBinOps = {"+", "-", "/", "*", "%"};
STMLTextTokens ABinOps = {"add", "sub", "div", "mul", "mod"};

STMLTextTokens CUnPrefixOps = {"++", "--", "!", "~"};
STMLTextTokens AUnPrefixOps = {"preinc", "predec", "not", "bnot"};
STMLTextTokens CUnPostOps = {"++", "--"};
STMLTextTokens AUnPostOps = {"postinc", "postdec"};

STMLModel Expr {

  STMLLeaf C_binops {

        range (i = 0,4,1) {
            input = {"$CBinOps[i]"};
            output = {"$ABinOps[i]"};
        };
   };

  STMLLeaf C_pre_uops {
          range (i = 0,3,1) {
                input = {"$CUnPrefixOps[i]"};
                output = {"$AUnPrefixOps[i]"};
          };

};

STMLLeaf C_post_uops {

          range (i =0,1,1) {
                input = {"$CUnPostOps[i]"};
                output = {"$AUnPostOps[i]"};
          };

};

STMLNode C_binary_expr {

          C_binops op;
          STMLAny o1, o2;
          input = { o1, op, o2 };
          output = { op, o1, o2 };
};

STMLNode C_prefix_unary_expr {

          C_pre_uops op;
          STMLAny o1;
```

```
            input = { op, o1 };
            output = { op, o1 };
    };

    STMLNode C_postfix_unary_expr {

            C_post_uops op;
            STMLAny o1;

            input = { o1, op };
            output = { op, o1 };
    };
};
```

Note:
For more information on creating a model file, please refer section 1.7 of this manual.

Step-2: Create the input file

Contents of array.in file:

```
E + E E - E E * E E / E E % E ++ E -- E ! E ~ E E ++ E --
```

Step-3: Invoke the translator as follows.

```
% st -m array.md array.in
```

On executing the above command, the contents of the input file are translated as per the model description to an equivalent content in the output representation.

The following output is displayed on the console.

```
add E E sub E E mul E E div E E mod E E preinc E predec E not E bnot E
postinc E postdec E
```

1.6.2 Sample using STML Client-Server Translator - ‘xml_text’

STML Client-Server Translator samples can be found under the directory $(STF_HOME)/samples/stml. The ‘**xml_text**’ sample explained below, uses the xml_text.md file as the model description file and xml_text.in and text_xml.in as the input files.

Step-1: Create the model file as below.

Contents of xml_text.md:

```
STMLModel PO {

    STMLLeaf POHeader {
        input = { "<PO>" };
        output = { "POSTART" };
    };

     STMLLeaf POFooter {
        input = { "</PO>" };
        output = { "POEND" };
    };
```

```
STMLLeaf SNO {

    STMLAny sno;

    input = { "<SERIAL>", sno, "</SERIAL>" };
    output   = { sno };
};

STMLLeaf DES {

    STMLAny des;

    input = { "<DESCRIPTION>", des, "</DESCRIPTION>" };
    output   = { des };
 };

STMLLeaf UP {

    STMLAny up;

    input = { "<UNITPRICE>", up, "</UNITPRICE>" };
    output   = { up };
};

STMLLeaf QTY {

    STMLAny qty;
```

```
        input = { "<QUANTITY>", qty, "</QUANTITY>" };
        output   = { qty };
};

STMLLeaf RT {

        STMLAny rt;

        input = { "<ROWTOTAL>", rt, "</ROWTOTAL>" };
        output   = { rt };
};

 STMLLeaf PORow {

        SNO sno;
        DES des;
        UP up;
        QTY q;
        RT rt;

        input = { sno, des, up, q, rt };
        output = { sno, des, up, q, rt };
 };

STMLLeaf POTotal {

        STMLAny GrandTotal;

        input = { "<TOTAL>", GrandTotal, "</TOTAL>" };
        output = { "Total", GrandTotal };
    };

    STMLLeaf PORowseq : sequence (PORow);

    STMLNode POBody {

        POHeader h;
        POFooter f;
        PORowseq r;
        POTotal t;

        input = { "S", h, r, t, f };
        output = { "S", h, r, t, f };
    };

};
```

Step-2: Create the input files as follows.

Contents of xml_text.in file:

```
 S <PO> <SERIAL> 1 </SERIAL> <DESCRIPTION> item1
</DESCRIPTION> <UNITPRICE> 2 </UNITPRICE>
<QUANTITY> 4 </QUANTITY> <ROWTOTAL> 8 </ROWTOTAL>
<SERIAL> 2 </SERIAL> <DESCRIPTION> item2
</DESCRIPTION> <UNITPRICE> 2 </UNITPRICE>
<QUANTITY> 4 </QUANTITY> <ROWTOTAL> 8 </ROWTOTAL>
<SERIAL> 3 </SERIAL> <DESCRIPTION> item3
```

```
</DESCRIPTION> <UNITPRICE> 2 </UNITPRICE>
 <QUANTITY> 4 </QUANTITY> <ROWTOTAL> 8 </ROWTOTAL>
 <TOTAL> 24 </TOTAL> </PO>
```

Contents of text_xml.in file:

```
S POSTART 1 item1 2 4 8 2 item2 2 4 8 3 item3 2 4 8 Total 24 POEND
```

Step-3: Invoke the STML Client-Server translator as follows.

```
% ns --VaradhiPORT 5040

% stml_server -DS <ip_addr> -DSP  5040 &

% stml_client -ORBInitRef NameService=<ip_addr>:5040 -m
   <absolute_path>/xml_text.md <abs_path_infile>/xml_text.in

%stml_client -ORBInitRef NameService=<ip_addr>:5040 -ik output -ok input
   -m <absolute_path>/xml_text.md <abs_path_infile>/text_xml.in
```

where,

<ip_addr>	-	IP address of the host where NameServer is running.
<absolute_path>	-	Absolute path of the model file.
<abs_path_infile>	-	Absolute path of the input file.

On executing the above commands, the contents of the input file are translated as per the model description to an equivalent content in the output representation.

The following output is displayed on the console.

```
S POSTART 1 item1 2 4 8 2 item2 2 4 8 3 item3 2 4 8 Total 24 POEND

S <PO> <SERIAL> 1 </SERIAL> <DESCRIPTION> item1
</DESCRIPTION> <UNITPRICE> 2 </UNITPRICE> <QUANTITY> 4
</QUANTITY> <ROWTOTAL> 8 </ROWTOTAL> <SERIAL> 2 </SERIAL>
 <DESCRIPTION> item2 </DESCRIPTION> <UNITPRICE> 2
</UNITPRICE> <QUANTITY> 4 </QUANTITY> <ROWTOTAL> 8
</ROWTOTAL> <SERIAL> 3 </SERIAL> <DESCRIPTION> item3
</DESCRIPTION> <UNITPRICE> 2 </UNITPRICE> <QUANTITY> 4
</QUANTITY> <ROWTOTAL> 8 </ROWTOTAL> <TOTAL> 24 </TOTAL>
 </PO>
```

1.7 Creating a simple application

1.7.1 Creating a model file

STML can be used to model hierarchical information such as data formats, message formats, protocol formats, language grammars etc. The following are the steps involved in the creation of a STML description of the information that is to be modeled and translated using STF. It has been explained using an example which converts a tagged document to a plain document.

Step 1: Identify the structure of the information being modeled.

Step 2: Identify the various representations needed to translate/transform the information.

Step 3: Create a model description file with .md extension.

Step 4: Add the following construct in the new file created above.

```
STMLModel <name>
{

};
```

where,

<name> is an identifier using which STF identifies this model.

Step 5: Identify the elements of the hierarchy and classify them as follows

Root elements	-	elements that occur at the top of the hierarchy
Internal node elements	-	elements that occur at the intermediate level in the hierarchy.
Leaf elements	-	elements that occur at the bottom of the hierarchy.

Step 6: For each Leaf element identified in the previous step do the following.

(i) Define a STMLLeaf structure for the element within the scope of the STMLModel as follows

```
STMLModel Model {

        STMLLeaf <name> {
        variable_declaration;
        rep1 = {...};
```

```
                rep2 = {...};
                ...
                repn = {...};

                };

        };
```

Here, <name> identifies the element (unique name should be used for each specific element that is described).

The <variable_declaration> section can be used to declare any variables which may be references to other STMLElements or standard types like STMLWord, STMLAny etc. The fields 'rep1', 'repn' etc refer to the different representations of the element.

(ii) Define each representation for the element as follows

a. Identify the sequence of values which make up this element in the modeled domain.

b. Use the following constructs to represent each of the values.

quoted string	- to represent constant string of characters.
STMLWord	- to represent a pre-defined sequence of characters.
STMLAny	- to represent any sequence of characters not including white space
STMLSymbol	- to represent a symbol which can be added to a symbol table.

STMLTextTokens	- to represent the predefined array of values.
range	- to represent a range of values either over a variable or an array.
union	- to specify alternate possibilities for a node in the element hierarchy.
sequence	- to specify the repeated occurrences of a node in the element hirearchy.

Example:

```
 STMLTextTokens tags = { "menu", "menubar", "toolbar" };

  STMLModel tag_to_plain_text {

      STMLLeaf Tag {

           range (i = 0, 2, 1) {
                       input =  { "$tags[i]" };
                       output = { "$tags[i]" };
          };

      };

      STMLLeaf Attr {

          STMLWord attr_name;
          STMLWord attr_value;

          input = { attr_name#"="#attr_value };
          output = { attr_name#"="#attr_value};
```

```
        };

        STMLLeaf Seq_Attr : sequence (Attr,6);

        STMLLeaf Ip_Tag_with_attr {

            Tag tag_name_attr;
            Seq_Attr att1;

            input = { "<"# tag_name_attr, att1# ">"};
            output = { "<"# tag_name_attr, att1# ">"};
        };

        STMLLeaf  Ip_Tag {

            Tag tg;

            input = { “<“#tg#”>”};
            output= { “<“#tg#”>” };
        };

        STMLLeaf st_tag_info : union ( Ip_Tag, Ip_Tag_with_attr) {};

    };
```

Here the tags are predefined in the form of the 'tags' array and a range declaration is used in 'Tag' to represent the array of values.

Note: ‘STMLTextTokens’ needs to be defined outside the STML Model.

Step 7: For each internal Node element identified in the earlier step do the following.

i) Define a STMLNode structure for the element within the scope of the STMLModel as follows

```
STMLModel Model {

  STMLNode <name> {

    variable_declaration;

    rep1 = {...};
    rep2 = {...};
      ...
    repn = {...};
  };

};
```

Here <name> identifies the element (unique name should be used for each specific element that is described).

The <variable_declaration> section can be used to declare any variables which may be references to other STMLElements or of standard types like STMLWord, STMLAny etc. The fields 'rep1', 'rep2' etc refer to the different representations of the element.

ii) Define each representation for the element as follows.

a. Identify the sequence of values which make up this element in the modeled domain.

b. Use the following constructs to represent each of the values:

quoted string	- to represent constant string of characters.
STMLWord	- to represent a pre-defined sequence of characters.
STMLAny	- to represent any sequence of characters not including white space.
STMLSymbol	- to represent a symbol which can be added to a symbol table.
range	- to represent a range of values either over a variable or an array.
identifier	- reference to a previously declared STMLNode or STMLLeaf element.
union	- to specify alternate possibilities for a node in the element hierarchy.
sequence	- to specify the repeated occurrences of a node in the element hierarchy.

Example:

```
STMLNode text_node {

        st_tag_info st_tag;
        Ip_Tag end_tag;
        STMLWord text;
```

```
            input = { st_tag, text, end_tag };
            output = { text };
};
```

The above example defines a node element called 'text_node' whose 'input' representation is composed of a st_tag_info (Ip_tag or Ip_tag_attr), some text and an end tag. Note the reference to the already declared (in earlier example) STMLLeaf element 'st_tag_info' and ‘Ip_Tag’.

Step 8: For each Root element identified in the previous step do the following.

(i) Define a STMLRoot structure for the element within the scope of the STMLModel as follows.

```
  STMLModel Model {

     STMLRoot <name> {

            variable_declaration;

             rep1 = {...};
             rep2 = {...};
             ...
             repn = {...};
     };
  };
```

Here <name> identifies the element (unique name should be used for each specific element that is described).

The <variable_declaration> section can be used to declare any variables which may be references to other STMLElements or of standard types like STMLWord, STMLAny etc. The fields 'rep1', 'rep2' etc refer to the different representations of the element.

(ii) Define each representation for the element as follows.

a. Identify the sequence of values which make up this element in the modeled domain.

b. Use the following constructs to represent each of the values.

quoted string	- to represent constant string of characters.
STMLWord	- to represent a pre-defined sequence of characters.
STMLAny	- to represent any sequence of characters not including white space.
STMLSymbol	- to represent a symbol which can be added to a symbol table.
range	- to represent a range of values either over a variable or an array.
identifier	- reference to a previously declared STMLRoot, STMLNode or STMLLeaf element.
union	- to specify alternate possibilities for a node in the element hierarchy.
sequence	- to specify the repeated occurrences of a node in the element hierarchy.

Example:

```
  STMLRoot Document {
          text_node node;
          input =   { node };
          output = { node };
  };
```

The above example shows a root element called 'Document' whose 'input' representation is composed of a text node (see previously defined STMLNode text_node). The 'output' representation is composed of the text.

The entire contents of the model description file for the tagged document to plain document conversion would be as follows.

```
STMLTextTokens tags = { "menu", "menubar", "toolbar" };

STMLModel tag_to_plain_text {

     STMLLeaf Tag {

          range (i = 0, 2, 1) {
               input =  { "$tags[i]" };
               output = { "$tags[i]" };
          };
     };

     STMLLeaf Attr {
```

```
    STMLWord attr_name;
    STMLWord attr_value;
    input = { attr_name#"="#attr_value };
    output = { attr_name#"="#attr_value};
};

STMLLeaf Seq_Attr : sequence (Attr,6);

STMLLeaf Ip_Tag_with_attr {

    Tag tag_name_attr;
    Seq_Attr att1;

    input = { "<"# tag_name_attr, att1# ">"};
    output = { "<"# tag_name_attr, att1# ">"};
};

STMLLeaf  Ip_Tag {

    Tag tg;
    input = { “<“#tg#”>”};
    output= { “<“#tg#”>” };
};

STMLLeaf st_tag_info : union ( Ip_Tag, Ip_Tag_with_attr) {};

STMLNode text_node {
```

```
            st_tag_info st_tag;
            Ip_Tag end_tag;
            STMLWord text;

            input = { st_tag, text, end_tag };
            output = { text, “\n” };
        };

        STMLRoot Document {
            text_node node;
            input =   { node };
            output = { node };
        };

    };
```

The above model file ‘test.md’ is used for translating a tagged document to plain document. The input file test.in contains the following.

```
<menu> Demo <menu>
<toolbar  type=data> HELLOWORLD <toolbar>
```

Note:

‘#’ operator known as paste operator, can be used to specify multiple tokens that are not delimted by white space, whereas ’,’ operator is used to specify the tokens delimited by white space.

1.7.1.1 Invoking STML Line Translator

The translation can be performed by invoking '*st*' from the command line as follows.

```
% st -m test.md test.in
```

On executing the above command, the following output will be displayed on the screen.

```
Demo
HELLOWORLD
```

1.7.1.2 Invoking stml_server and stml_client

Varadhi Naming Service 'ns', STML Server 'stml_server' and STML Client 'stml_client' applications can be run from any host. By default, the 'stml_server' listens on port number 3222. To change this default port number, use the -PORT option.

On Unix csh Prompt,

```
% ns --VaradhiPORT 5040

% stml_server -DS <ip_addr> -DSP  5040 &

%stml_client -m <absolute_path>/test.md <abs_path_infile>/test.in
  -ORBInitRef NameService=<ip_addr>:5040
```

where,

<ip_addr>	-	IP address of the host where NameServer is running.
<absolute_path>	-	Absolute path of the model file.
<abs_path_infile>	-	Absolute path of the input file.

On executing the above commands, the following output will be displayed on the console.

```
Demo
HELLOWORLD
```

REFERENCE MANUAL

Sankhya Technologies Private Limited

Part 2 - Reference Manual

2.1 STML Line Translator - st

2.1.1 Introduction

STML Line Translator is a model driven translation and transformation tool that can both parse information (document, message, data) in one representation and automatically translate the information to any other representation based upon the specified STML model.

For reversible transformations, ***'st'*** can recreate the original document, message or data from the transformed document. This is extremely useful when converted documents are edited and the original document needs to be updated with the changes.

2.1.2 Synopsis

```
st -m mdfile [-ik kind]   [-ok kind] [-prop file] [-p model] [-sym file]
[-t match_type] [-stream "name:lib ..."] [-v] [-V] [-h] input_file
```

2.1.3 Description

STML Line Translator translates the input as per the model file 'mdfile' (a .md file) and displays the output in the target format. The input can be specified in a file input_file or in command line.

2.1.4 Options

The following are the options supported by ‘st’ .

-ik kind	This option is used to specify the input representation for translation. The default value is ‘input’. Supported values are ‘input’, ‘output’.
--ok kind	This option is used to specify the output representation for translation. Supported values are ‘input’ and ‘output’. The default value is ‘output’.
-prop file	This option is used to specify the property description file for translation.
-p model	This option is used to specify the model to use for translation.
-sym file	This option is used to specify the symbol file from which symbol is to be loaded
-t match_type	This option is used to specify the matching element type. Supported values are ‘any’, ‘root’, ‘node’ and ‘leaf’. The default value is ‘node’.

-stream “name:lib...”	This option is used to specify the stream library to be used for stream ‘name’. Multiple library specifications can be provided as a space separated list.
-v	This options is used to specify that ‘st’ needs to be invoked in verbose mode.
-V	This option is used to display the version information of the translator.
-h	This option will display the help message.

2.2 STML Client-Server Translator - stml_server

2.2.1 Introduction

STMLServer is a CORBA based document server that can be used to transform or translate documents from one format to another based upon the specified Sankhya Translation Modeling Language (STML) model(s) in a distributed environment.

Any CORBA compliant client can access the STMLServer in a distributed environment. STMLServer also includes database (DB) stream support, model stream support, logical data base (LDB) support, input file stream support and directory stream support.

2.2.2 Synopsis

```
% stml_server [-log] [ [-DS <ip_addr>] [-DSP <ds_port> ] ] [-PORT port]
                    [-cfg <path>] [-s]
```

2.2.3 Description

STML Server translates the input as per the model file ‘.mdfile’ and sends the output in the target format to the client. STML Server provides operations for specifying the document information (model file, input file, property model file, STMLModel, session, symbol information, input and output representation) so that any CORBA compliant client can access the server for translation.

STML server checks for the configuration file, stml_server.cfg, in the following locations in the given order and reads the configuration parameters from the file.

a) The path specified by -cfg option in the command-line.

b) $STF_HOME/etc (%STF_HOME%\etc on Windows)

c) The directory from which stml_server was started.

Each line in the configuration file specifies a parameter name and its value. Lines beginning with '#' are treated as comments and are ignored.

2.2.4 Options

The following are the options supported by ‘stml_server’.

-DS <ip_addr>	-	This option is used to specify the IP address where naming service is running.
-DSP <ds_port>	-	This option is used to specify the port in which naming service is listening,

-PORT <port>	-	This option is used to specify the port in which stml_server will be listening,
-log		This option enables the logging of server messages to a log file. The messages are written to a file named ‘stml_server.log'. The location of this file is selected as follows: - If STF_LOG_FILE_PATH_PREFIX environment varable is set then the log file is created under this path. - Otherwise, the log file is created under the directory from which stml_server was invoked. If the log file already exists in the specified path then further log messages are appended to it.
-cfg <path>		This option specifies the path to the directory where the configuration file stml_server.cfg is present. Note that '<path>' should not include the configuration file name.
-s		This option causes STML server to operate in single-process mode. All client requests will be handled by a single server process.

2.2.5 STML server - Multi-Process Mode

STML Server operates in multi-process mode by default. New sessions can be started in child processes. Each child process can handle a specified number of sessions after which it is terminated. The total number of child processes, total number of sessions and maximum sessions per process are configurable. The configuration parameters can be specified through the configuration file, 'stml_server.cfg'.

The following parameters are used to control the multi-process server.

a) max_process:

The maximum number of child processes that can exist at any time. If max_process is set to zero, then stml_server will operate in single-process mode. The hard limit on max_process is 256. Default value is 50.

b) min_process:

The minimum number of child processes that need to be available at any time. If min_process is greater than max_process, it is set to max_process. This setting is ignored in singe-process mode. Default value is 5.

c) max_session:

The maximum number of sessions that can be created at any time. For multi-process server, the upper limit on max_session is determined by the product of max_process and max_session_per_process. Default value is 500.

d) max_session_per_process:

The maximum number of simultaneous sessions per process. This setting is ignored in single-process mode. Default value is 10.

e) total_session_per_process:

The total number of sessions that a child process can handle during its lifetime. Once this limit is reached, the process is automatically terminated. This setting is ignored in single-process mode. Default value is 10.

For most installations, the default values for the above parameters should be sufficient. The maximum processes or sessions supported will depend on operating system limits and can vary from system to system. Specifying too high a value for any of the above parameters may cause stml_server to not function properly and may also impact other applications running on the system. If such a condition occurs, then try reducing the values or use default settings for the parameters.

Note:

1) The -s option causes the server to operate in single-process mode irrespective of the configuration file settings.
2) The multi-process server support is currently available on Linux host only. On other hosts, stml_server runs in single-process mode only.

2.3 STML Client-Server Translator - stml_client

2.3.1 Introduction

STML_Client is a CORBA based client which can communicate with the STMLServer to translate or transform documents in a distributed environment.

2.3.2 Synopsis

```
stml_client -ORBInitRef NameService=<ip-addr>:<port> [options] -m model-file input-file
```

2.3.3 Description

STML_Client takes a STML model file ('model-file') and input file ('input-file') as inputs and provides them to the STML Server for translating the input as per the model. It receives the output from the server and displays it to the standard output. The stml_client uses CORBA technology to communicate with the server which could be running on the same host as the client or on a different host in a network.

Note: The model-file and input-file names should be absolute pathnames and all the files required for a translation (model, input, symbol and property file) should be accesible by the STML Server.

2.3.4 Options

-m model-file	This option is used to specify the model file to be used for translation.
-p model	This option is used to specify the name of the model to be used for the translation.
-prop file	This option is used to specify the property description file to be used for translation.
-sym file	This option is used to specify the file from which symbols used for translation are to be loaded.
-ik kind	This option is used to specify the input representation for translation. Supported values for 'kind' are 'input' (default) and 'output'.
-ok kind	This option is used to specify the output representation for translation. Supported values for 'kind' are 'input' and 'output' (default).

The following CORBA ORB options should be specified always:

-ORBInitRef NameService=<ip-addr>:<port>

This option is used to specify the location and port number of the CORBA NameServer.

<ip-addr> refers to the IP address of the host (in a.b.c.d notation) on which the Name Server is running and

<port> refers to the port on which the Name Server accepts client connections.

2.4 SANKHYA Translation Modeling Language

SANKHYA Translation Modeling Language (STML) is a simple yet powerful language for modeling document formats, message formats and language structures.

STML allows multiple formats of data to be described in a single model description. This feature is used to build model driven tools, which translates and transforms one format to any other format, described using STML. This finds application in Enterprise Application Integration, Program translation tools (like code generators, assemblers, binary translator) and Natural Language Translation tools.

STML allows the specification of the hierarchies of structured information and the description of each node in hierarchy in several different representations (different languages, different data formats etc). It supports various data types for representing concepts of the input domain naturally.

STML also supports specification of input streams from which data has to be obtained during translation. This provides a powerful mechanism for obtaining input from different sources (file, ftp, database, directory etc) during translation process.

2.4.1 Notation

The following notations are used in the STML specification.

“string”	-	matches <string> literally
[c1-c2]	-	matches any of the characters between c1 and c2 (both inclusive)
[^list]	-	matches any character except those specified by ’list’
RE1 | RE2	-	matches RE1 or RE2
RE*	-	matches zero or more occurrences of RE
RE+	-	matches one or more occurrences of RE
(RE)	-	matches RE

2.4.2 Lexical Elements

The following is the list of lexical elements used in the STML specification.

D	-	[0-9]	#Decimal Digit
O	-	[0-7]	# Octal Digits
H	-	[0-9A-Fa-f]	# Hexadecimal Digits
B	-	[0-1]	# Binary Digits

STRING	-	'[^']*'	#Quoted String
INTEGER	-	(D)+ | "-"(D)+	#A Decimal Integer
HEX_NUMBER	-	"0x"(H)+ | "0X"(H)+	#A Hexadecimal Integer
BINARY_NUMBER	-	"0b"(B)+ | "0B"(B)+	#A Binary number
IDENTIFIER	-	([A-Z]|[a-z])(([A-Z]+)|([a-z]+)|([0-9_]+))*	#An Identifier

2.4.3 STML 1.0 Specification

1. stml_spec :=
(a) stml_model_list |
(b) stml_declaration_list

2. stml_model_list :=
(a) stml_model |
(b) stml_model stml_model_list

3. stml_model :=
(a) STMLModel model_name ’{’ element_list ’}’ ’;’ |
(b) STMLModel model_name : model_name ’{’ element_list ’}’ ’;’

4. model_name := STRING

5. element_list :=
(a) element |
(b) element element_list

6. element :=
(a) root_element ’;’ |
(b) node_element ’;’ |
(c) leaf_element ’;’

7. root_element :=
(a) stml_root element_name ’{’ element_description_list ’}’ |
(b) stml_root element_name : element_name ’{’ element_description_list ’}’
(c) stml_root element_name : ’union’ ’(’ union_name_list ’)’ ’{’ ’}’

8. element_description_list :=
(a) element_description |
(b) element_description element_description_list

9. element_description :=
(a) child_element_declaration |
(b) representation_list

10. child_element_declaration :=
element_type variable_list ';'

11. element_type :=
(a) stml_type |
(b) user_defined_type

12. stml_type :=
(a) STMLWord |
(b) STMLAny |
(c) STMLSymbol

13. user_defined_type := IDENTIFIER

14. variable_list :=
(a) variable |
(b) variable variable_list

15. variable := IDENTIFIER

16. node_element :=
(a) stml_node element_name ’{’ representation_list ’}’ |
(b) stml_node element_name : element_name ’{’ representation_list ’}’ |
(c) stml_node element_name : ’union’ ’(’ union_name_list ’)’ ’{’ ’}’

17. leaf_element :=
(a) stml_leaf element_name ’{’ representation_list ’}’ |
(b) stml_leaf element_name : element_name ’{’ representation_list ’}’ |
(c) stml_leaf element_name : ’union’ ’(’ union_name_list ’)’ ’{’ ’}’ |
(d) stml_leaf element_name ’{’ range_representation ’}’ |
(e) stml_leaf element_name : sequence '{' element_name '}' |
(f) stml_leaf element_name : sequence '{' element_name, INTEGER '}'

18. stml_root := ’STMLRoot’
19. element_name := STRING

20. representation_list :=
(a) representation |
(b) representation representation_list

21. union_name_list :=
(a) IDENTIFIER |
(b) IDENTIFIER ’,’ union_name_list

22. stml_node := ’STMLNode’

23. stml_leaf := ’STMLLeaf’

24. representation := representation_name ’=’ ’{’ value_list ’}’ ’;’

25. range_representation := range_specification ’{’ representation_list ’}’

26. range_specification := ‘range’ ’(’ range_name ’=’ range_start ’, ’range_end’ ’,’ range_increment ’)’

27. range_name := IDENTIFIER

28. range_start := INTEGER

29. range_end := INTEGER

30. range_increment := INTEGER

31. representation_name := IDENTIFIER

32. value_list :=
(a) value |
(b) value ’,’ value_list

33. value :=
(a) string_value |
(b) [attributes] element_name [properties] |
(c) stream_value |
(d) binary_value

34. attributes := '{' attribute_list '}'

35. string_value := STRING

36. binary_value :=
(a) const_binary_value

37. const_binary_value :=
(a) ’or’ ’(’ INTEGER ’,’ INTEGER ’,’ const_value ’)’
(b) IDENTIFIER ’(’ INTEGER ’,’ INTEGER ’)’
(c) ’expr’ ’(’ STRING ’,’ INTEGER ’,’ INTEGER ’)
(d) ’$’ IDENTIFIER
(e) const_value

38. const_value :=
(a) STRING
(b) INTEGER
(c) HEX_NUMBER
(d) BINARY_NUMBER

39. attribute_list :=
(a) attribute |
(b) attribute ’,’ attribute_list

40. attribute := attribute_name ’=’ attribute_value_list

41. attribute_name := IDENTIFIER

42. attribute_value_list :=
(a) attribute_value |
(b) attribute_value ’:’ attribute_value

43. attribute_value := IDENTIFIER

44. stream_value :=
(a) 'stream' stream_name '{' value_list '}' |
(b) 'stream' properties stream_name '{' value_list '}'

45. stream_name := STRING

46. stml_declaration_list :=
(a) stml_declaration |
(b) stml_declaration stml_declaration_list

47. stml_declaration := array_declaration

48. array_declaration :=
(a) array_type array_name ’=’ ’{’ array_string_list ’}’ ’;’ |
(b) array_type array_name ’=’ ’{’ array_number_list ’}’ ’;’

49. array_type := IDENTIFIER
50. array_name := IDENTIFIER
51. array_string_list :=
(a) STRING |
(b) STRING ’,’ array_string_list

52. array_number_list :=
(a) INTEGER |
(b) INTEGER ’,’ array_number_list

53. properties := '{' property_list '}'

54. property_list :=
(a) property |
(b) property ',' property_list

55. property := IDENTIFIER '=' STRING

2.4.4 STMLModel

A STMLModel description contains a set of model specifications and optionally a list of data declarations. A stml_model is a description of any hierarchical data composed of a set of elements. The elements can be thought of making the nodes of the tree formed by the entity being modeled. Each model is given a name ’model_name’ which can be used to refer to a particular model.

Example:

```
 STMLModel MyModel
{
     ...
};
```

2.4.5 STML Element

Three kinds of elements can be described in STML - namely STMLRoot, STMLNode and STMLLeaf.

2.4.5.1 STMLRoot

STMLRoot represents the root or top-level node in the hierarchy of the structured entity being modeled. There can be multiple STMLRoot elements in a model representing different alternatives for the top-level node.

Each root_element is given a name, element_name using which it can be accessed. A root_element can be inherited from another element (Refer STML Specification No.7(b)), so that any properties of the parent element can be shared by the child. A root_element description consists of a list of declarations of child elements and a representation list. A child element of a root can be a STMLNode or STMLLeaf element or of a predefined STMLType.

A representation provides the description of an STML element in a particular format. A representation is composed of a representation_name which identifies the format and a list of values for the element in that particular format. A representation_list is a list of such representations. Each STML element can contain a representation_list. A ’representation’ is composed of a ’representation_name’ which identifies the format and a list of values for the element in that particular format. It provides the description of a STML element in a particular format. The value_list in a representation consists of a list containing strings/references to other STML elements which are child elements of the current element in the hierarchy.

Example:

```
STMLModel English {
      STMLRoot Sentence1
      {
        Subject s;
        Object o;
        input = {  s, o }; //Representation in English
      };
};
```

Here, the description of a ’root_element’ ’Sentence1’ is provided. It consists of references to two child elements, Subject (s) and Object (o). The ’representation_list’ consists of one ’representation’ - with name ’input’ - and provides the representation of Sentence1 in english as consisting of a subject and an object.

Example:

```
STMLModel English_to_French  {
      STMLRoot Sentence1        {
         Subject s;
         Object o;
         input = {  s, o }; //Representation in English
         output = {  s, o }; //Representation in French
      };
};
```

The above example is similar to the previous one except that the description of the root element “Sentence1” contains one more representation in french named ’output’.

2.4.5.2 STMLNode

STMLNode represents an element which can occur at the internal levels of the tree structured information. Each node_element is given a name, element_name using which it can be accessed. A node_element can be inherited from another element (Refer STML Specification No.16(b)), so that any properties of the parent element can be shared by the child.

A node_element descriptions consists of a list of declarations of child elements and a representation list. A child element of a STMLNode can be a STMLLeaf element or an instance of a predefined STMLType but not a STMLNode. A STMLNode element can have a representation_list similar to the one mentioned for a STMLRoot.

Example:

```
STMLModel English_to_French {
    STMLNode Subject {
        Noun n;
        input = { n };
        output = { n };
    };
    STMLRoot Sentence1 {
        Subject s;
        Object o;
        input = {  s, o };
        output = {  s, o };
    };
};
```

The example used in the STMLRoot section is expanded above to include the description of a STMLNode element named ’Subject’. Here Subject is shown to have a reference to another STML element named ’Noun’.

2.4.5.3 STMLLeaf

STMLLeaf represents an element which can occur at the bottom-level of the hierarchy. STMLLeaf does not exactly correspond to the traditional concept of leaf and nodes of a tree structure, since it is possible to specify the child elements for a STMLLeaf element. However, such child elements should themselves be either STMLLeaf elements or instances of predefined STML types.

Each leaf_element is given a name, element_name using which it can be accessed. A leaf_element can be inherited from another element (Refer STML Spec No. 17(b)), so that any properties of the parent element can be shared by the child. A leaf_element description consists of a list of declarations of child elements and a representation list. A child element of a STMLLeaf cannot be a STMLNode or a STMLRoot element. A STMLLeaf element can have a representation_list similar to the one mentioned for STMLRoot and STMLNode.

Example:

```
STMLModel English_to_French  {
     STMLLeaf Noun  {
          input = { "the" };
          output = { "le" };
     };
     STMLNode Subject  {
          Noun n;
          input = { n };
          output = { n };
     };
     STMLRoot Sentence1
     {
          Subject s;
          Object o;

          input = {  s, o };
          output = {  s, o };
     };

};
```

The previous example under STMLNode has been extended to include a STMLLeaf element named ’Noun’. The representation_list for this element includes string values for input (english) and output (french) representations.

2.4.6 STML Representations

A STML representation models the structured information in a particular format. The format could be a data format (XML), a language (assembly language, natural language), a message format (SOAP) etc.

STML allows the specification of multiple representations for the entity being modeled. That is, each element in the hierarchy of the modeled system can be described as different representations. For e.g., a document structure can be described as XML, HTML and text formats at the same time. This allows the equivalence between these formats to be specified and also allows translations of input data between these formats.

For each representation that is of interest in a domain, a specification should be provided while describing STML elements. The specification captures the syntax of the element in each particular representation. Consider the example model English_to_French described earlier. Here, ’input’ and ’output’ form the two representations and for each STML element (Sentence1, Subject, Noun), the structure of the element is specified in both the representations as a list of STMLValues.

2.4.7 STML Range Representation

It is possible to specify a range of values for a STML element using the range representation format.

Example:

```
STMLLeaf register
{
        range (i = 1,30,1)
       {
        asm = { "r$i" };
        mcode = { “$i” };
        };


};
```

Here a set of values for the STMLLeaf element ’register’ is specified using the range syntax. The “asm” (assembly) representation specifies the set of values: “r1”, “r2”, “r3”, ... “r30” and the “mcode” (machine code) representation specifies a set of values : 1, 2, 3, ... 30. The variable ‘i’ is used to represent the range of values from 1 to 30 with an increment of 1.

2.4.8 STML Value

A STML Value is a string, binary value or a reference to a STML element.

String Value

Example:

```
STMLRoot Sentence
{           Subject s;
            Object o;
            input = {  "English", s, o };
            output = {  "French", s, o };
};
```

In the above example, "English" and "French" are string values and s, o are references to STML elements Subject and Object respectively.

Binary Value

Example:

```
 STMLNode add
{
        register rs,rt,rd;
        asm = {"add", rd, rs, rt };
         mcode = { or(0,6,0b100000); rd(6,5); rs(11,5); rt(16,5); or(21,11,0)};

};
```

The example above shows, the use of binary values. The representation ’mcode’ provides the machine code for an ’add’ instruction as a list of binary values. Each value specifies a start bit, length and value to be placed at that bit position. Note that element references like ’rd(6,5)’ specify only the start bit and length of the value. The actual value will be supplied by the child element (i.e., register). Binary values specified in ‘mcode’ needs to be separated by ‘;’ .

A Binary value can be of the following form:

(a) ’or’ ’(’ INTEGER ’,’ INTEGER ’,’ const_value ’)’

Example:

```
or(0,6,"100000");
or(0,6,32);
or(0,6,0x20);
or(0,6,0b100000);
```

(b) IDENTIFIER ’(’ INTEGER ’,’ INTEGER ’)’

Example:

```
rd(6,5);
```

(c) ’expr’ ’(’ STRING ’,’ INTEGER ’,’ INTEGER ’)’

(d) ’$’ IDENTIFIER

Example:

```
STMLLeaf register  {
        range (i = 1,30,1)
       {
          mcode = { “$i” };
       };
};
```

(e) const_value

```
mcode = { 1 };
mcode = { "1" };
mcode = { 0x1 };
mcode = { 0b1 };
```

2.4.9 STML Attributes

STML allows attributes to be associated with STML Values. An STML Attribute can be used to specify the properties of string, reference and binary values. An STML Attribute is a name-value pair. Attributes can be used to differentiate between similar elements during translation.

Syntax : { attr-list }

where,

attr-list - name=value [,...].

Any set of characters within “{ }” is taken to be an attribute in the model. By default, attributes are turned off while processing. To process attributes in the input, process_attributes property should be set.

Example:

```
STMLNode add
{
    register rs,rt,rd;
    asm = {"{width=32}add", rd, rs, rt };
    mcode = { or(0,6,0b100000); rd(6,5); rs(11,5); rt(16,5);   or(21,11,0)};
};
```

Attributes are specified within { } bracket pairs before a string, binary or reference. In the example above, a ’width’ attribute has been specified for the “add” instruction and its value has been specified as ’32’. This allows the selection of this STMLNode to be controlled based on the width attribute. A tool can check for the ’add’ instruction along with the width attribute’s value to select this particular element.

Multiple attributes can be specified for a value as follows:

```
{sign=s, width=32, type=int:ptr} value
```

Each attribute in the list is separated by a comma. Multiple values for an attribute are specified using a ’:’ separated list as shown for ’type’ attribute above.

Following are the ways to modify the above behaviour by setting the properties.

- process_attr

When the property 'process_attr' is set to 'on', { occuring in the input will be considered as an attribute specification. The above property if 'off' will make the translator not to process any attributes in the input.

- attr_start and attr_end

  Example:

  ```
  attr_start <
  attr_end >
  ```

When the properties 'attr_start' and 'attr_end' is set to different characters other than '{' and '}' respectively, the specified characters will act as attribute delimiters for the translator.

- match_attr

When the property 'match_attr' is set to off, attributes will be processed in input but matching will not be done with model attributes.

Note:

1. Set ‘process_attr’ to ‘on’, if the attributes needs to be processed.

2. Set 'attr_start' and 'attr_end' to different characters, if the attributes needs to be specified in model files and the input can contain '{' and '}'.

3. Set 'match_attr' to off, if the model file and input will contain attributes but matching them is not needed.

2.4.10 STML Union

The STML ‘union’ feature can be used to specify alternate possibilities for a node in the element hierarchy.

Example:

```
STMLModel Tag {
    STMLLeaf Item {
      ...
      ...
    };

    STMLLeaf Quantity {
      ...
      ...
    };

    STMLLeaf any : union ( Item, Quantity ) {};

    STMLNode all {
      any a;
      input = { a };
      output = { a };
    };
};
```

In the above example, the STML union construct is used to accept either 'Item' or 'Quantity' as valid leaf nodes.

2.4.11 STML Sequence

The STML 'sequence' feature is used to specify the repeated occurrences of a node in the element hierarchy. The types of sequence are bounded sequence and unbounded sequence.

2.4.11.1 UnBounded Sequence

UnBounded sequence is used to specify any number of repeated occurrences of a node. The input will be matched till the match occurs for the sequence element type.

Syntax:

```
    STMLLeaf sequence_name : sequence ( element_name );
```

where,

element_name	-	name of already defined element of type STMLLeaf or the name of the union of STMLLeaf's.

Example:

```
    STMLModel Comment_Sequence {
        STMLLeaf newline {
          ...
          ...
        };
        STMLLeaf ws {
          ...
```

```
            ...
          };
          STMLLeaf Comment {
            ...
            ...
          };
          STMLLeaf Token : union ( newline, ws, Comment ) {};
          STMLLeaf TokenList : sequence ( Token );
          STMLNode TokenStream {
              TokenList s;
              input = { s };
              output   = { s };
          };

     };
```

2.4.11.2 Bounded Sequence

Bounded sequence is used to specify the specified number of repeated occurrences of a node. The input will be matched for the specified number of times of the sequence element type.

Syntax:

```
    STMLLeaf sequence_name : sequence ( element_name, INTEGER );
```

where,

element_name	-	name of already defined element of type STMLLeaf or the name of the union of STMLLeaf's.
INTEGER	-	number of repeated occurrences of a node in the element hierarchy.

The second parameter in the sequence construct is optional. If the second parameter is unspecified or specified as zero the input is matched until the match occurs for the sequence element type. If the second parameter is specified as ’num’, then the input will be matched ’num’ times.

Example:

```
STMLModel Comment_Sequence {
    STMLLeaf newline {
      ...
      ...
    };
    STMLLeaf ws {
      ...
      ...
    };
    STMLLeaf Comment {
      ...
      ...
    };
    STMLLeaf Token : union ( newline, ws, Comment ) {};
    STMLLeaf TokenList : sequence ( Token, 5 );
    STMLNode TokenStream {
      TokenList s;
      input = { s };
      output   = { s };
    };
};
```

In the above example, the STML sequence construct is used to accept the sequence of either ’newline’ or ’ws’ or ’Comment’ as valid leaf nodes.

2.4.12 STML Types

STML supports the following types, instances of which can be used in element definitions.

2.4.12.1 STMLWord

STMLWord represents any sequence of characters beginning with an alphabetic character (a-z or A-Z) and followed by any number of alphanumeric characters. Word properties can be specified for each of the STMLWord to specify the acceptable characters in the beginning, middle and in the end of the word using ‘**wbegin**’, ‘**wmid**’ and ‘**wend**’ attributes in the input representation.

Example:

```
STMLNode tc_record {
        STMLWord date;
        input =  { date {wbegin="0-9", wmid="0-9/"},
        output = { “Date:”, date };
};
```

In the above example, ‘date’ is defined as an STMLWord. The properties *wbegin* and *wmid* are defined for ‘date’ to include only those specific characters in the beginning and in the middle of the word respectively. Usage of any other characters other than the ones defined, will give an error message. The error message will provide the input file name, position where the processing of input failed and the failed token.

2.4.12.2 STMLAny

STMLAny represents any space delimited sequence of characters.

Example:

```
STMLLeaf Website {
  STMLAny a;

  input = { a };
  output = { a };

};
```

2.4.12.3 STMLSymbol

STMLSymbol is similar to STMLAny but can be installed in a symbol table and can be looked-up.

2.4.13 STML Properties

STML allows properties to be defined separately to delimit the acceptable characters in the input. The following lists the properties and their descriptions in a model file.

Property Name	: **wbegin**
Description	: Specifies the characters that can be used to begin a STMLWord
Value	: Any regular expression
Default	: [a-zA-Z]

Property Name	: **wmid**
Description	: Specifies the characters that can occur at positions other than the beginning of a STMLWord
Value	: Any regular expression
Default	: [a-zA-Z0-9]

Property Name	: **wlen**
Description	: Specifies the length of the word to be read from input
Value	: Non-negative number
Default	: None

Property Name	: **process_attr**
Description	: Indicates if attributes in the input should be processed or not
Value	: on, off
Default	: off

Property Name	: **match_attr**
Description	: Indicates if attributes in the input should be matched with the model attributes
Value	: on, off
Default	: on

Property Name	: **attr_start**
Description	: Attribute start specifier
Value	: Any Character
Default	: '{'

Property Name	: **attr_end**
Description	: Attribute end specifier
Value	: Any Character
Default	: '}'

Property Name	: **skipws**
Description	: Indicates if white-spaces should be skipped in the input

Value	: true, false
Default	: true

Property Name	: **union_match_type**
Description	: Specifies whether a union match should terminate after first matching element is found or if it should continue till the best match is found.
Value	: first, best
Default	: first

Property Name	: **v_output**
Description	: Specifies whether the corresponding element should be considered for output. If the value is "true" then it is included in the output, otherwise it is excluded.
Value	: true, false
Default	: true

Property Name	: **v_match**
Description	: Specifies whether the corresponding element should be considered for output. If the value is "true" then it is included in the match, otherwise it is excluded.
Value	: true, false
Default	: true

Following describes the sample properties file and their way of invocation using STF.

Example:

```
wbegin [a-zA-Z]
wmid [a-zA-Z_:.]
wend [a-zA-Z0-9]
match_case false
skipws true
union_match_type best
```

‘st’ can be invoked as follows for the above property file ‘prop1.txt’ .

```
% st -m test.md test.in -prop prop1.txt
```

where,

test.md	-	model file
test.in	-	Input file
prop1.txt	-	property description file

‘stml_server’ can be invoked using the following steps.

```
%ns --VaradhiPORT 5060 &

%stml_server  -DS <ip_addr>  -DSP 5060 &

%stml_client -ORBInitRef NameService=<ip_addr>:5060
    -m <absolute_path>/test.md <abs_path_infile>/test.in -prop prop1.txt
```

where,

absolute_path	-	Absolute path of the model file ‘test.md’
abs_path_infile	-	Absolute path of the input file ‘test.in’
test.md	-	model file
test.in	-	input file
ip_addr	-	IP Address of the host where naming service is running.

Following is the contents of another property description file ‘prop2.txt’ that indicates to skip the white space in the input.

```
skipws false
```

Following property description file ‘prop3.txt’ contains the property description for setting the attribute delimiters.

```
process_attr on
attr_start <
attr_end >
```

2.4.14 STML Stream

During translation, certain applications require inputs to be read from multiple streams (of different types) and written to different output streams at different point of time. To support this, STF supports the specification, creation and use of different types of input streams.

STML Stream construct allows the specification of input streams from or to which data should be obtained or sent during translation. The format for stream specification is as follows:

stream [properties] STRING '{' value_list '}'

where,

STRING is a quoted string of the following form,

stream-id:path

where,

stream-id	-	string which identifies the stream type. Example: ’file’, ’str’, ’symbol’, ’ftp’ etc.
path	-	specifies the location of the stream source or destination. Example: ’path’ could be the path name for file stream, ftp site location for ftp stream etc.

'properties' is an optional list of properties for the stream
(please refer to item 53 in section 2.4.3)

The following are the different types of streams that are supported by STF.

2.4.14.1 Symbol stream

In Symbol stream, the input word to be translated is obtained from the value of the symbol whose name is specified in the model. Default value for the symbol can be specified in the model file. Following are the three different syntax and their description behaviour for symbol stream.

Syntax1: "sym:$(symbol:default-value)"

Example:

```
    input = { "translate", w , stream "sym:$(MYSYMBOL:german_text)"
                    { d } };
```

The symbol table will be checked first and if the symbol is not found in the symbol table the default value specified in the model file (i.e., german_text) will be assigned to the symbol. But the symbol table will not be updated.

Syntax2: "sym:$(symbol:=default-value)"

Example:

```
   input = { "translate", w , stream "sym:$(MYSYMBOL:=german_text)"
                { d } } ;
```

The symbol table will be checked first and if the symbol is not found in the symbol table the default value specified in model file will be assigned to the symbol. The symbol table will also be updated with the symbol name and it's default value.

Syntax3: "sym:$(symbol=default-value)"

Example:

```
input = { "translate", w , stream "sym:$(MYSYMBOL=german_text)"
            { d } };
```

The default value specified in the model file will be assigned to the symbol. The symbol table will also be updated with the symbol name and it's default value.

Note:

There should not be any space in the symbol stream string syntax.

Following is the complete STMLLeaf description of a symbol stream using the syntax1.

```
STMLLeaf xlate {
   STMLWord w;
   TokenStream d;

   input = { "translate", w, stream "sym:$(MYSYMBOL:german_text)"   {d}};
   output = { d };
};
```

As shown in the example, the value list of representation 'input for the STMLLeaf element 'xlate' contains a stream declaration

stream "sym:$(MYSYMBOL:default_value)" {d}

The above indicates that whenever the word 'translate' is seen in the input the input stream should be changed to that specified by the symbol "MYSYMBOL", if the symbol value is found in the symbol table or to the default value "german_text". All subsequent inputs that match TokenStream 'd' (another STML element) will be obtained from the new stream. Once the input from the stream is exhausted the new stream will be closed and input will be obtained from the previous stream. The input word to be translated is obtained from the value of the symbol whose name is specified in the model. Note that some external mechanism is required to install the symbol "MYSYMBOL" with appropriate stream inputs.

2.4.14.2 String stream

In String stream, the input word to be translated is obtained from the string specified in the model.

Example:

```
STMLLeaf xlate2 {
    tr d;
    input = { "translate", stream "str:Mercadoria" { d } };
    output = { d };
};
```

The above example indicates that whenever the word ’translate’ is seen in the input, the input stream is changed to that specified by the string. All subsequent inputs that match TokenStream ’d’ (another STML element) are obtained from the new stream. Once the input from the stream is exhausted, the new stream is closed and input is obtained from the previous stream.

2.4.14.3 File Stream

The input word to be translated is obtained from a file whose location is specified in the model using the STML ’stream’ construct. File stream will be picked up from the location in the environment variable STF_FS_PATH_PREFIX or the directory from which the translation tool ‘st’ or ‘stml_server’ is invoked.

Example

```
STMLLeaf tc_one
{
        tr d;
        input = { "translate", stream "file:stream.txt" { d } };
        output = { d };
};
```

The above example indicates that whenever the word ’translate’ is seen in the input, the input stream is obtained from the file mentioned in the model. All subsequent inputs that match TokenStream ’d’ (another STML element) are obtained from the new file stream. Once the input from this stream is exhausted, the new file stream is closed and the input is obtained from the previous stream.

2.4.14.4 Database Stream

The input to be translated is obtained from the database. The translator will process and validate the database attributes based on the Data Source Name (DSN).

ODBC Connectivity with the database is established using the Data Source Name (DSN), User name, Password specified in the input file. Sql query in the input file is then executed and the output from the database is used for translation.

Example

```
STMLLeaf table1Start {

        STMLWord w1;
        STMLWord w2;
        STMLWord w3;
        STMLWord w4;

        TableTokenList d;

        input  = { "Table", "src=db",
        "dsn="#w1{wbegin="A-Za-z0-9",  wmid="a-zA-Z0-9-._"},
        uid="#w2{wbegin="A-Za-z0-9",  wmid="a-zA-Z0-9-._"},
        "pwd="#w3{wbegin="A-Za-z0-9",wmid="a-zA-Z0-9-._"},
        "query=\""#w4{wbegin="A-Za-z", wmid="a-zA-Z0-9.-*,;_/ "}# "\"",
         stream "db:DSN=$w1;UID=$w2;PWD=$w3;:$w4" {d} };
        output = { d };

    };
```

As shown in the example, the value list of representation input for the STMLLeaf element 'table1start' contains a stream declaration

stream "db:DSN=<dsn_name>;UID=<user_name>;

PWD=<password>:sql_query {d}

The input data to be translated is obtained from the database. The above example indicates that whenever data in the following format is seen in the input, the input stream is changed to that specified by the output of the database execution query. All subsequent inputs that match TokenStream 'd' (another STML element) are obtained from the new stream. Once the input from the stream is exhausted, the new stream is closed and input is obtained from the previous stream.

2.4.14.4.1 Database stream output:

The default output of the database stream will be in the following format:

```
<START>                          ---> Table start tag
  <CNS> field names <CNE>         ---> Header providing field names
  <RS>                           ---> Row start tag
       <CS>                      ---> Field (column) start tag
         field value              ---> Field value from database
       <CE>                      ---> Field end tag
       <CS>
         field value
        <CE>
         ...
  <RE>                           ---> Row end tag
  <RS>
         ...
 <RE>
 ...
<END>                            ---> Table end tag
```

The different tags used in the database stream output can be modified by providing the following properties in the stream specification.

TABLE 1. **Database Stream Properties**

Property Name	Description	Value	Default
db_ts	Table start tag	String	START
db_te	Table end tag	String	END
db_fns	Start tag for field name list header	String	CNS
db_fne	End tag for field name list header	String	CNE
db_rs	Row start tag	String	RS

TABLE 1. **Database Stream Properties**

db_re	Row end tag	String	RE
db_fs	Field (column) start tag	String	CS
db_fe	Field end tag	String	CE
db_thdr	controls output of field name list header	on/off	on

The following special keywords can be used in the property value string to control the output. These keywords are replaced in the output as described below.

TABLE 2. **Special Keywords**

Keyword	**Description**
#field-name	Replaced with the name of the field
#field-number	Replaced with the sequential number of the field
#row-number	Replaced with the sequential number of the row

Example:

The following example illustrates the use of database stream properties to control the format of the output obtained for a query.

```
stream { db_ts="table", db_te="/table",
      db_rs="row", db_re="/row",
      db_fs="#field-name", db_fe="/#field-name",
      db_thdr="off"
     }
  "ldb:ldb_src=db_xml;:select * from table" { d } };
```

The above causes the table start tag to be set to "table" and the end tag to "/table". Also, the row start and end tags are set to "row" and "/row" respectively. Note the use of the keyword #field-name in the field start and end tags. This will cause the names of the fields from the database table to be used as the field start and end tags instead of the default (CS and CE respectively). Finally, the display of the field name list is turned off by specifying db_thdr as "off".

The output for any query obtained using the above stream specification will be in the following format:

```
<table>
  <row>
    <name> ... </name>
     ...
  </row>
  ...
</table>
```

where, 'name' corresponds to the actual name of the field in the table. For eg, if the result contains only one field named 'AccountNumber', then the output will be as follows:

```
<table>
  <row>
    <AccountNumber> ... </AccountNumber>
  </row>
  ...
</table>
```

2.4.14.5 Logical Database Stream

The input to be translated is obtained from the database. The translator will process and validate the database attributes based on the logical DSN.

ODBC Connectivity with the database is established using the Data Source Name (DSN), User name, Password specified in the initialization file. Sql query in the input file is then executed and the output from the database is used for translation. The initialization file and the sql query needs to be specified in the input file.

Example

```
STMLLeaf table1Start {

        STMLWord w1;
        STMLWord w2;
        TableTokenList d;

        input  = { "TABLE", "src=db", "ldb_src="# w1
        {wbegin="A-Za-z0-9",wmid="a-zA-Z0-9-._"},
         "query=\""#w2{wbegin="A-Za-z", wmid="a-zA-Z0-9.-*,;_/ "} #"\"",
         stream "ldb:ldb_src=$w1;:$w2" {d} };

        output = { d };
};
STMLLeaf dbStart {

        STMLWord w1;
        STMLWord w2;

        PassTokenList d;

        input  = { "DATABASE", "ldb_src="#w1
        {wbegin="A-Za-z0-9", wmid="a-zA-Z0-9-._"},
         "table=\""#w2{wbegin="A-Za-z", wmid="a-zA-Z0-9.-*,;_"}#"\"",
         stream "ldb:ldb_src=$w1;:Table=$w2" {d} };
        output = { d };
};
```

As shown in the example, the value list of representation input for the STMLLeaf element 'table1start' contains a stream declaration

stream "ldb:ldb_src=<ini_file>:sql_query {d}

The value list of representation input for the STMLLeaf element 'dbStart' contains a stream declaration

stream "ldb:ldb_src=<ini_file>:Table=<table_name> {d}

The input data to be translated is obtained from the database. The above example indicates that whenever data in the following format is seen in the input, the input stream is changed to that specified by the output of the database execution query. All subsequent inputs that match TokenStream 'd' (another STML element) are obtained from the new stream. Once the input from the stream is exhausted, the new stream is closed and input is obtained from the previous stream.

For Logical Database Stream output please refer to the section 2.4.14.4.1

2.4.14.6 Model Stream

The input document will be processed and translated using sequential stml models, where the output of intermediate translation will serve as the input for next translation.

Example

```
//xml_text.md
//All rights reserved.

STMLModel PO {
      STMLLeaf POHeader {
        input  = { "<PO>" };
        output = { "POSTART" };
};
STMLLeaf POFooter {
        input  = { "</PO>" };
        output = { "POEND" };
};
STMLLeaf SNO {
        STMLAny sno;
        input  = { "<SERIAL>", sno, "</SERIAL>" };
        output = { sno };
};
STMLLeaf DES {
        STMLAny des;
        input  = { "<DESCRIPTION>", des, "</DESCRIPTION>" };
        output = { des };
};
STMLLeaf UP {
        STMLAny up;
        input  = { "<UNITPRICE>", up, "</UNITPRICE>" };
        output = { up };
};
STMLLeaf QTY {
        STMLAny qty;
        input  = { "<QUANTITY>", qty, "</QUANTITY>" };
        output = { qty };
};
STMLLeaf RT {
        STMLAny rt;
        input  = { "<ROWTOTAL>", rt, "</ROWTOTAL>" };
        output = { rt };
};

STMLLeaf PORow {
        SNO sno;
        DES des;
        UP up;
        QTY q;
        RT rt;

        input  = { sno, des, up, q, rt };
        output = { sno, des, up, q, rt };

};

STMLLeaf POTotal {
        STMLAny GrandTotal;

        input  = { "<TOTAL>", GrandTotal, "</TOTAL>" };
        output = { "Total", GrandTotal };
};

STMLLeaf PORowseq : sequence (PORow);

STMLNode POBody {
        POHeader h;
        POFooter f;
        PORowseq r;
```

```
                POTotal t;
            input  = {  h, r, t, f };
            output = { "S", h, r, t, f };
        };
    };
```

'**st**' needs to be invoked as follows for the above sample.

```
% st -m xml_text.md xml_text.in | st -m xml_text.md -ik output -ok input text_xml.in
```

'**stml_server**' needs to be invoked as follows for the above sample.

```
        % ns --VaradhiPORT 5050 &

        % stml_server -DS <ip_addr> -DSP 5050 -PORT 4567 &
```

for XML_text conversion:

```
        % stml_client -ORBInitRef NameService=<ip_addr>:5050
           -m <absolute_path1>/xml_text.md  <abs_path_infile1>/xml_text.in ,
```

for text_XML conversion:

```
      % stml_client -ORBInitRef NameService=<ip_addr>:5050 -ik output -ok input
          -m <absolute_path1>/xml_text.md <abs_path_infile2>/text_xml.in
```

where,

ip_addr	-	IP Address of the system where naming service is running.
absolute_path1	-	Absolute path of the model file 'xml_text.md'
abs_path_infile1	-	Absolute path of the input file ‘xml_text.in’
abs_path_infile 2	-	Absolute path of the input file ‘text_xml.in’

The input file xml_text.in is processed and the output (text_xml.in) translated using the representation in the model file xml_text.md will serve as the input for the next translation performed using the same model file xml_text.md .

2.4.14.7 Directory Stream

The input to be translated is obtained from the directory whose location is specified in the model using the STML ’stream’ construct. Using directory stream, the files and directories in the specified directory can be listed. The directory will be picked up from the location pointed out by the environment variable STF_DIR_PATH_PREFIX or the directory from which the translation tool ‘stml_server’ is invoked.

Example:

```
STMLLeaf dir_leaf {

      TokenStream d;
      input = { "translate", stream "dir:test" { d } };
      output = { d };
};
```

As shown in the example, the value list of representation ’input’ for the STMLLeaf element ’dir_leaf’ contains a stream declaration

stream "dir:test" {d}

The above indicates that whenever the word ’translate’ is seen in the input, the input stream should be changed to the specified directory stream ’test’ . All subsequent inputs that match TokenStream ’d’ (another STML element) will be obtained from the new directory stream ‘test’. Once the input from the stream is exhausted, the new stream will be closed and the input will be obtained from the previous stream.

The following lists the model file ’dir_stream.md’ using directory stream.

```
STMLModel Test_dir_stream
{

    STMLLeaf passthrough {

        STMLAny a;
        input = { a };
        output = { a };
    };

    STMLLeaf nl {

        STMLAny a;
        input = { a#"\n" };
        output = { a#"\n" };
    };

    STMLLeaf ws {

        input = { " " };
        output = { " " };
    };

    STMLLeaf TokenStream;

    STMLLeaf dot {
```

```
    input = {"DIRSTA", "." ,"DIREND"};
    output = { "" };
};

STMLLeaf dotdot {

    input = { "DIRSTA", ".." , "DIREND"};
    output = { "" };
};


STMLLeaf dircont {

    STMLWord t;
    input = { "DIRSTA", t{wbegin="a-zA-Z0-9"} , "DIREND"};
    output = { t#"/" };
};

STMLLeaf TokenStream1;

STMLLeaf dir_leaf{

    TokenStream1 d;
    input = { "translate", stream "dir:test" { d } };
    output= { d };
};

STMLLeaf null {
        input = { "" };
        output   = { "" };
};

STMLLeaf Dirstart {

        input = {"STARTDIR"};
        output = { "\n" };
};

STMLLeaf Filestart1 {

        STMLWord fil;
        input  =  {"FILESTA",  fil{wbegin="A-Za-z0-9",  wmid="a-zA-Z0-9Ž-
_",

         wend="a-zA-Z0-9"}, "FILEEND" };
        output = { fil };
};

STMLLeaf Dirend {

        input = {"ENDDIR"};
        output = { " " };
};

STMLLeaf Token1 : union ( passthrough, nl, ws, Dirend, Filestart1, dotdot,
dot, Dirstart, dir_leaf, dircont  ) {};
```

```
STMLLeaf TokenList1;

STMLLeaf TokenStream1 {

        Token1 f;
        TokenList1 n;
        input = { f, n };
        output   = { f, n };

 };

STMLLeaf TokenList1 : union (null, Token1, TokenStream1) {};

 STMLLeaf null {
        input = { "" };
        output   = { "" };
};

STMLLeaf Token : union ( passthrough, nl, ws, dir_leaf  ) {};

STMLLeaf TokenList;

STMLLeaf TokenStream {
        Token f;
        TokenList n;
        input = { f, n };
        output   = { f, n };
 };
```

```
        STMLLeaf TokenList : union (null, Token, TokenStream) {};

        STMLNode Document {
                TokenStream sp;
                input = { sp };
                output   = { sp };
         };

    };
```

The input file ‘dir_stream.in’ for the above model file will contain the following.

```
translate
```

’**stml_server**’ needs to be invoked as follows for the above sample.

```
%ns --VaradhiPORT 5060 &

%stml_server -DS <ip_addr> -DSP 5060 -PORT 5678 &

%stml_client -ORBInitRef NameService=<ip_addr>:5060
  -m <absolute_path>/dir_stream.md <abs_path_infile>/dir_stream.in
```

where,

ip_addr	- IP Address of the system where naming service is running.
absolute_path	- Absolute path of the model file ’dir_stream.md’ .
abs_path_infile	- Absolute path of the input file ‘dir_stream.in’.

2.4.15 STML Declarations

STML allows declaration of text and numeric arrays of values which can be used in STML element descriptions.

Example:

```
STMLTextTokens english_question = { "who", "when", "why" };
STMLTextTokens french_question = { "qui", "quand", "pourquoi" };

STMLModel English_to_French
{
    STMLLeaf question
    {
            range (i = 0, 2, 1)
           {
                input = { "$english_question[$i]" };
                output = { "$french_question[$i]" };
           };
    };
};
```

As shown above, two arrays of strings ’english_question’ and ’french_question’ have been defined. These arrays have been used in the definition of the STMLLeaf element ’question’. So, the element ’question’ can match any value from the arrays ’english_question’ and ’french_question’ for english and french representations respectively. Case sensitive match from the arrays 'english_question'and 'french_question' will be done by the translation tool.

Appendix A - SANKHYA Varadhi ™

The **SANKHYA Translation Framework** Client-Server Edition uses CORBA® technology for client-server interaction. A CORBA-compliant middleware is required for the Client-Server Edition to operate. This includes a CORBA Object request Broker (ORB) and a Naming Service component.

STF uses **SANKHYA Varadhi** as the CORBA middleware. SANKHYA Varadhi is Sankhya's object middleware solution for cross platform distributed systems development. Varadhi provides software developers, the tools and components required to develop distributed software applications that can run either within an organization's Intranet or across the Internet.

Varadhi enables distribution of software across Windows, Linux and Solaris hosts and embedded systems like mobile phones and PDAs. Varadhi manages the diversity in programming languages, location and host computers thereby enabling application developers to concentrate on implementing the application logic.

For more information on SANKHYA Varadhi, please refer to the following web page:

http://www.sankhya.com/info/varadhi.html

For obtaining SANKHYA Varadhi, please refer to the following web page:

http://www.sankhya.com/info/products/varadhi/download.html

For SANKHYA Varadhi Documentation, please refer to the following web page:

http://www.sankhya.com/info/products/varadhi/docs.html

INDEX

For More Information-

STF Download	http://www.sankhya.com/info/products/data/download.html
STF Documentation	http://www.sankhya.com/info/products/data/docs.html
STF Sales & Support	sales@sankhya.com

SANKHYA™

Sankhya Technologies Private Limited
#13/2, "JayaShree", Third Floor, First Street, Jayalakshmipuram, Nungambakkam,
Chennai 600 034, INDIA
Tel: +91 44 2822 7358
Fax: +91 44 2822 7357

Sankhya Technologies India Operations Private Limited
#30-15-58,"Silver Willow",Third Floor,
Dabagardens
Visakhapatnam 530 020, INDIA
Tel:+91 891 554 2666
Email: sales@sankhya.com
http://www.sankhya.com
