Universal Multiscript Layout Engine for Complex Text Layout Scripts

March 24, 1999

Chookij Vanatham
Prabhat Hegde
Ienup Sung
Sun Microsystems, Inc.

Contents

- Overview of CTL (Complex Text Layout) languages
- Solaris CTL support framework
- UMLE (Universal Multiscript Layout Engine) architecture and implementation
- Demonstration
- Comparison and concluding remarks
- Q&A

Page 2

Each of *CTL (Complex Text Layout)* languages like Arabic, Hebrew, Hindi, Thai, and so on requires a special processing before actual rendering of characters on display devices through a software component we call *layout engine (LE)* that is usually dedicated for one script/writing system.

In this presentation, we will present a single, universal layout engine that can be used for all CTL scripts by using the *Unicode Bidirectional Algorithm* [Unicode Technical Report #9] and a finite state machine-based shaping algorithm with user-supplied shaping rules. Specifically, the following will be presented:

- Overview of CTL languages.
- Solaris CTL support framework that makes possible creating and supporting CTL languages in Solaris system.
- A single, universal LE for all CTL scripts that we call *UMLE (Universal Multiscript Layout Engine)*.
- Demonstration of the UMLE.
- A brief comparison between UMLE approach and language-specific LE approach and concluding remarks.

Complex Text Layout Languages (1)

- Certain language's writing system requires special processing before actual rendering of text:
  - Bidirectional scripts (Arabic, Hebrew, Farsi, …)

    اللغة العربية لغة جميلة
  - Context-sensitive shaping scripts (Thai, Arabic, Devanagari, …)

Page 3

In the languages of the world especially based on Latin, Cyrillic, and, Greek scripts, there is no difference between how text is stored for data processing and how it is presented on a display device or a printer in usual cases. The text is read on horizontal lines from left to right, the lines progress from top to bottom and the characters are stored in a manner identical to how they are processed.

However, not all the languages of the world share these characteristics. Especially, some Middle East and South East Asian languages like Arabic, Hebrew, Thai, Lao, Hindi, and so on, are such exceptions that require a special processing of character/text data before actual rendering of the characters/text as they have following two peculiar characteristics in their text layout:

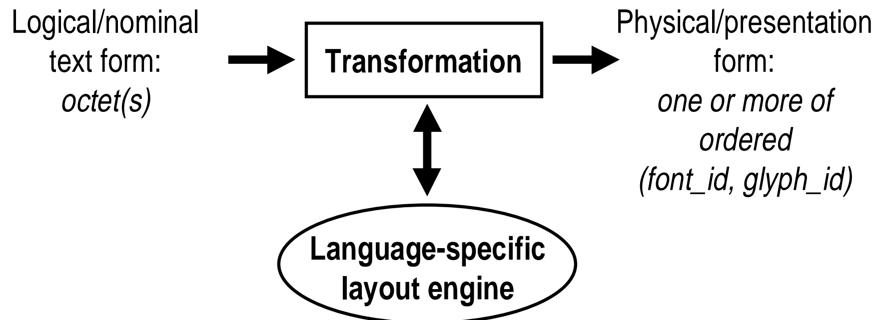
- Bidirectionality:

  Usually characters are written from right to left with some portions of text written from left to right, such as numbers and embedded Latin-based characters.
- Context-sensitive shaping:

  Usually character changes its shape and location within a rendering area column of the character(s), i.e., a display cell, depend on the location and/or surrounding characters. E.g., an Arabic character can have four different shape forms.

Complex Text Layout Languages (2)

- Pre-processing before actual rendering:



Page 4

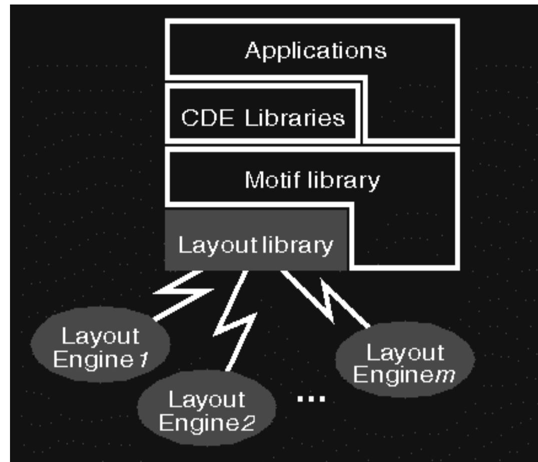
Such characteristics, bidirectionality and context-sensitive shaping, are language/script specific and differently inherent to each and every individual code points of the codesets of the scripts.

Due to such characteristics of bidirectionality and context-sensitive shaping, it is required to do a special pre-processing before actual rendering/presentation of characters, i.e., reordering and shaping, also known as, *transformation*.

Since these characteristics are language/script-specific and differently inherent to each and every individual character, it was, up until now, quite difficult to perform the special pre-processing before actual rendering with a single unified algorithm in a computer program. Therefore, usually, such special processing has been done within a software component that we call *layout engine* (LE) which is specifically dedicated to a single language/script.

The diagram in this slide shows an example of possible transformation scenarios.

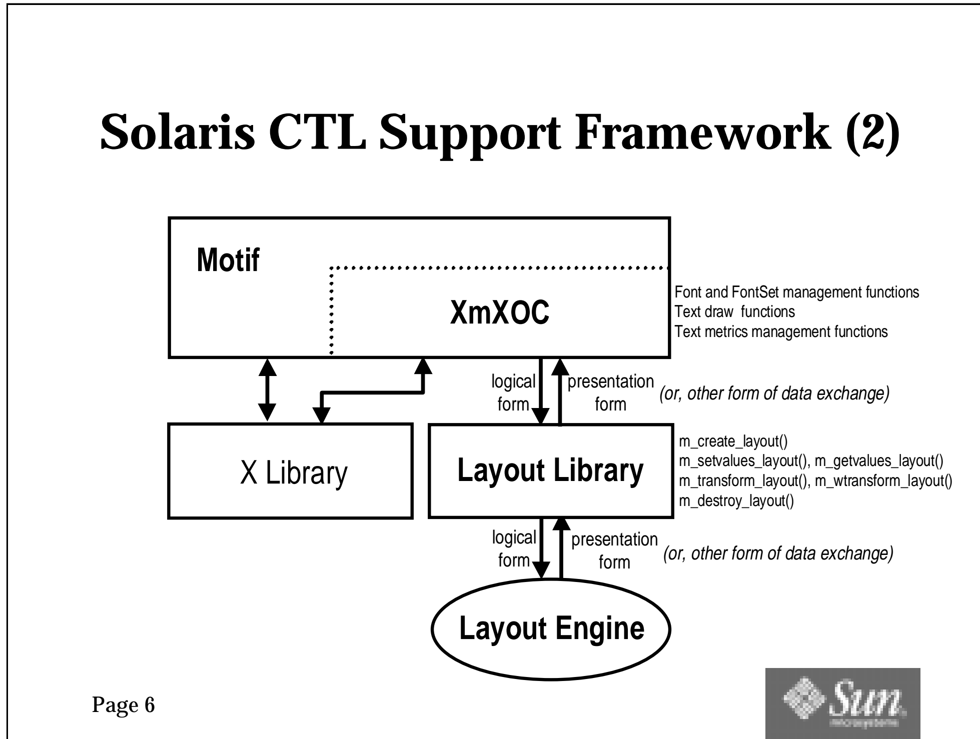
Solaris CTL Support Framework (1)



Page 5

Solaris CTL support framework consists of three main components:

- *Motif library* that uses the layout services of the layout library to provide complex text layout capabilities transparently to the toolkit users,
- *Layout library* that conforms to *X/Open CAE Specification -- Portable Layout Services: Context-dependent and Directional Text* and contains six internationalization programming interfaces for CTL language support, and,
- *Layout engines* that will be dynamically loaded/plugged into the layout library depend on the locale that the application program is running with.



In more detail, the three components interact as depicted in the diagram:

• From the Motif toolkit layer, when any of static/dynamic text object widget is created, the toolkit dynamically loads the layout library and then checks if the current locale requires CTL processing or not by calling `m_create_layout`(3X) and `m_getvalues_layout`(3X) functions. When the `m_create_layout`(3X) function is called, the current locale's LE is loaded/plugged into the layout library.

• If the current locale does not require the CTL processing, the widget does not go through the pre-processing steps of CTL but directly uses X library's output functions like `XmbDrawString`(3X11) without any pre-processing.

• If the current locale requires the CTL processing, however, the widget tries to get necessary layout values from the layout engine by calling `m_getvalues_layout`(3X) for the future processing. Afterwards, all of text rendering goes through an additional step of transformation before the actual rendering by using `m_transform_layout`(3X) or `m_wtransform_layout`(3X) functions. For the actual rendering, X library output functions like `XmbDrawString`(3X11), `XDrawString`(3X11), `XDrawString16`(3X11), and so on are used.

Detail description on each component will be presented in the next slides.

Solaris CTL Support Framework (3)

- Motif library:
  - Transparently modified to support complex text layout scripts.
  - All text object widgets and gadgets support the complex text layout.
  - CTL enabled in the new Motif 2.1 library, **libXm.so.4**, at Solaris 7.

Page 7

Motif and upper layer software components have transparently modified to support the CTL scripts in the Solaris operating environment. As a result, properly internationalized CDE/Motif applications including all CDE Desktop applications will support the CTL scripts without any code change.

The applications, however, have to be *re-linked* to the new, CTL-enabled Motif library, i.e., *libXm.so.4*, since Solaris 7 provides both libXm.so.3 (OSF/Motif 1.2.x based) and libXm.so.4 (OSF/Motif 2.1 based), and the default Motif library for previous releases of Solaris systems was the libXm.so.3.

For more detail on Motif library's CTL support internals, refer to *Complex Text Layout Support in Solaris Motif* paper in this proceedings.

Solaris CTL Support Framework (4)

- Layout library:
  - Conforms to *X/Open CAE Specification -- Portable Layout Services: Context-dependent and Directional Text.*
  - Contains total six public APIs:
    - `m_create_layout`(3X) / `m_destroy_layout`(3X)
    - `m_getvalues_layout`(3X) / `m_setvalues_layout`(3X)
    - `m_transform_layout`(3X) / `m_wtransform_layout`(3X)
  - Can be linked to any application by using "`-llayout`" option during compilation.

Page 8

The layout library in Solaris is a Sun implementation of *X/Open CAE Specification -- Portable Layout Services: Context-dependent and Directional Text.*

It contains total six APIs that can be used by any service users of the CTL service. Within Solaris, these interfaces are currently being used by two major software components:

• Motif library (libXm.so.4) for CTL support in GUI programming.

• `ctlmp`(1) printing filter that converts and beautifies given text files to PostScript files with proper CTL processing.

Detail information on each programming interfaces will be presented at slide page 13 to 18.

Since the library is a dynamic library, a.k.a., a shared object library, it can be loaded into an application at run time by using `dlopen`(3X) or directly linked to the binary at compile time by using "`-llayout`" option.

Solaris CTL Support Framework (5)

- Layout library (cont'd):
  - Manages Layout Object population.
  - It is the pluggable layer where a locale specific layout engine is dynamically loaded/plugged into.
  - Locales that have a layout engine in Solaris:
    - Arabic (Egypt) locale (`ar`) with ISO 8859-6 and Unicode 2.1 based font.
    - Hebrew locale (`he_IL`) with ISO 8859-8 font.
    - Thai locale (`th_TH`) with TIS 620.2533 based font.
    - All Unicode locales with above and more fonts.

Page 9

One of the main roles of the layout library is to manage the population of the *Layout Objects (LO)*. The LO is an opaque data type that encapsulates necessary information for the transformation.

The `m_create_layout`(3X) function returns a LO. Other layout library interfaces require the LO as an input argument since the LO contains not only the necessary information for the transformation but also some other internal states that need to be kept between the layout library function calls.

During the creation of the LO, the layout library loads the current locale's layout engine into the current process' memory space if it has not yet loaded and maps internal function tables for the six APIs of the library to the ones from the layout engine. Except for the population management, the layout library is nothing but a wrapper to a locale-specific layout engine, all of the layout services functionalities actually coming from the layout engine.

Solaris CTL Support Framework (6)

- Layout engine:
  - Layout engine is a program that provides locale-specific layout services to upper layer software for one or more locales.
  - Provided services:
    - Layout Object creation and destruction.
    - Setting and getting of layout values/attributes.
    - Layout transformations of input text such as reordering, text shaping, numeral shaping, swapping, and so on.

Page 10

Each layout engine is a program that provides a particular locale-specific layout services to upper layer.

Following are provided services:

- Layout object creation and destruction.
- Setting and getting of layout values/attributes.
- Layout transformations of input text such as reordering, text shaping, national numeral shaping, swapping, and so on.

Detail information on each of above services will be described in slide page 13 to 18.

UMLE (1)

- UMLE is a layout engine that is *codeset independent*, *user-customizable*, and, *multiscript capable* such that it can handle any kind of scripts in providing the layout services.
- Because it is universal, i.e., codeset independent, user-customizable, and, multiscript capable, it can handle not only Unicode but also any other kind of codesets.

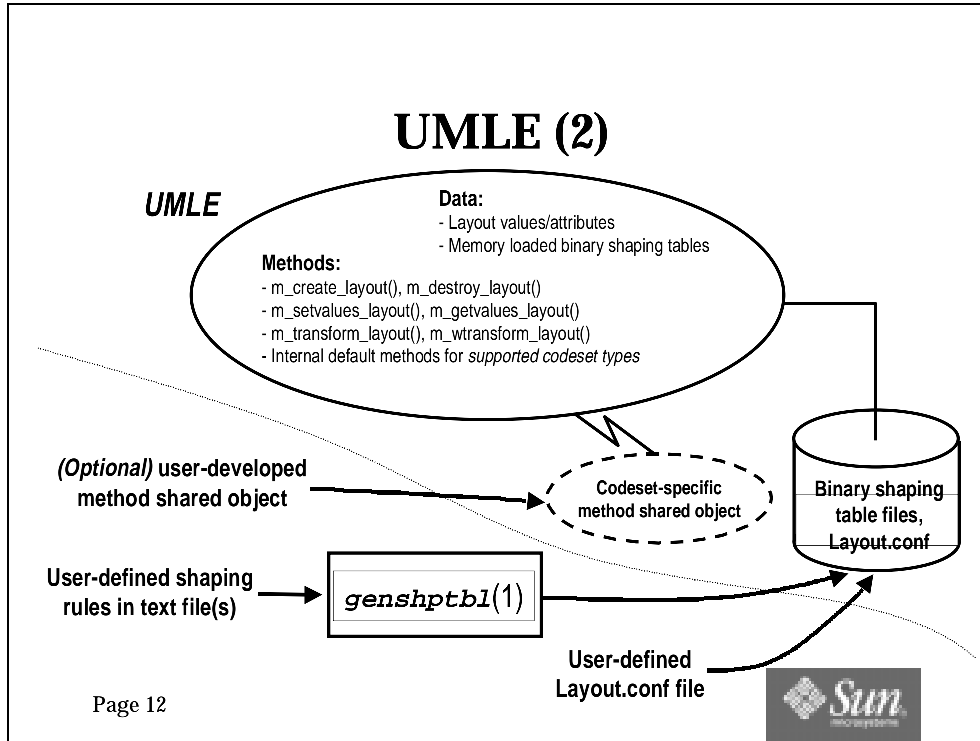
Page 11

UMLE is a layout engine that has designed and implemented to meet following three goals:

• It should be *codeset independent* so that it can be used for any kind of CTL script support in any kind of codeset.

•It should be *user-customizable* so that users can create a new behavior or, can customize existing behavior of the layout engine without touching the source of the layout engine.

• It should be able to handle *multiple scripts.*



To achieve aforementioned three goals, we defined following principles in the design of the UMLE:

- Use of codeset independent methods either from the UMLE pre-defined method sets or user-supplied method shared object.
- Use of Unicode Bidi Algorithm described in Unicode Technical Report #9 for bidirectionality support.
- Devise a binary table-driven shaping algorithm for context-sensitive shaping support.
- Develop a binary shaping rule table generator utility, namely, `genshptbl`(1), so that users can use the utility to generate the binary shaping rule table that the shaping algorithm will load into the UMLE.
- The configuration information will be provided by user in flat text file format so that the configuration information will be loaded into the UMLE in the beginning of the UMLE instantiation.

The result of the design is depicted in this slide. Details on each components with the layout services programming interfaces will be presented from slide page 13 to 19.

UMLE (3)

- m_create_layout(3X):
  - Load necessary binary shaping tables and default layout values into data section of memory by parsing the configuration file (Layout.conf).
  - Optionally, dynamically load and link user-defined external codeset-specific methods.
  - Create and initialize the Layout Object.

Page 13

The m_create_layout(3X) function of the UMLE creates a LO, reads the configuration file, Layout.conf, and loads, necessary shaping tables, locale's layout values/attributes, and, codeset-specific method designation definitions.

After that, the function initializes the LO and returns the LO.

If any error occurs during the procedure, the function returns (LayoutObject)0 and sets errno to indicate the error. For more detail on the errors, refer to m_create_layout(3X) man page.

UMLE (4)

- `m_setvalues_layout`(3X) and `m_getvalues_layout`(3X):
  - Can be used to set (S) or get (G) layout values from the Layout Object:

Layout value descriptor	Type	SG
Orientation	LayoutTextDescriptor	SG
Context	LayoutTextDescriptor	SG
TypeOfText	LayoutTextDescriptor	SG
ImplicitAlg	LayoutTextDescriptor	SG
Swapping	LayoutTextDescriptor	SG
Numerals	LayoutTextDescriptor	SG
TextShaping	LayoutTextDescriptor	SG
ActiveDirectional	BooleanValue	G
ActiveShapeEditing	BooleanValue	G
ShapeCharset	char *	SG
ShapeCharsetSize	int	G
ShapeContextSize	LayoutEditSize	G
InOutTextDescrMask	unsigned long	SG
InOnlyTextDescr	unsigned long	SG
OutOnlyTextDescr	unsigned long	SG
CheckMode	LayoutDesc	SG
QueryValueSize	int	G

Page 14

By using `m_getvalues_layout`(3X) and `m_setvalues_layout`(3X) functions, users can set or get layout values/attributes associated with the corresponding LO.

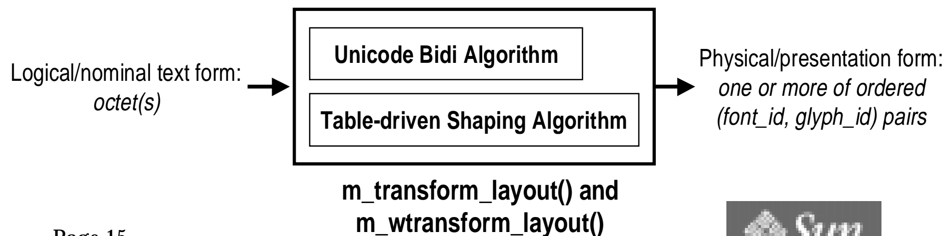
Some of the layout values are only possible to get, e.g., `ActiveDirectional`, `ActiveShapeEditing`, and so on, since they are defined by the layout engine cannot be altered. `ActiveDirectional` is a descriptor shows whether reordering of directional code elements are required or not and `ActiveShapeEditing` is another descriptor defines whether context-sensitive shaping is required or not in this LO.

In case of the UMLE, all of the default values for the layout value descriptors come from the user-supplied `Layout.conf` file. If `TypeOfText`, `Swapping`, `Numerals`, `TextShaping` and so on changes by using `m_setvalues_layout`(3X) function, the corresponding shaping rule tables will be used in the subsequent `m_transform_layout`(3X) function call(s).

For more detail on each layout value descriptors, refer to *X/Open CAE Specification -- Portable Layout Services: Context-dependent and Directional Text*.

UMLE (5)

- `m_transform_layout`(3X):
  - Uses Unicode Bidirectional Algorithm of UTR#9 for bidirectionality support.
  - Uses `genshptbl`(1)-generated shaping tables and Sun-proprietary table-driven shaping algorithm.



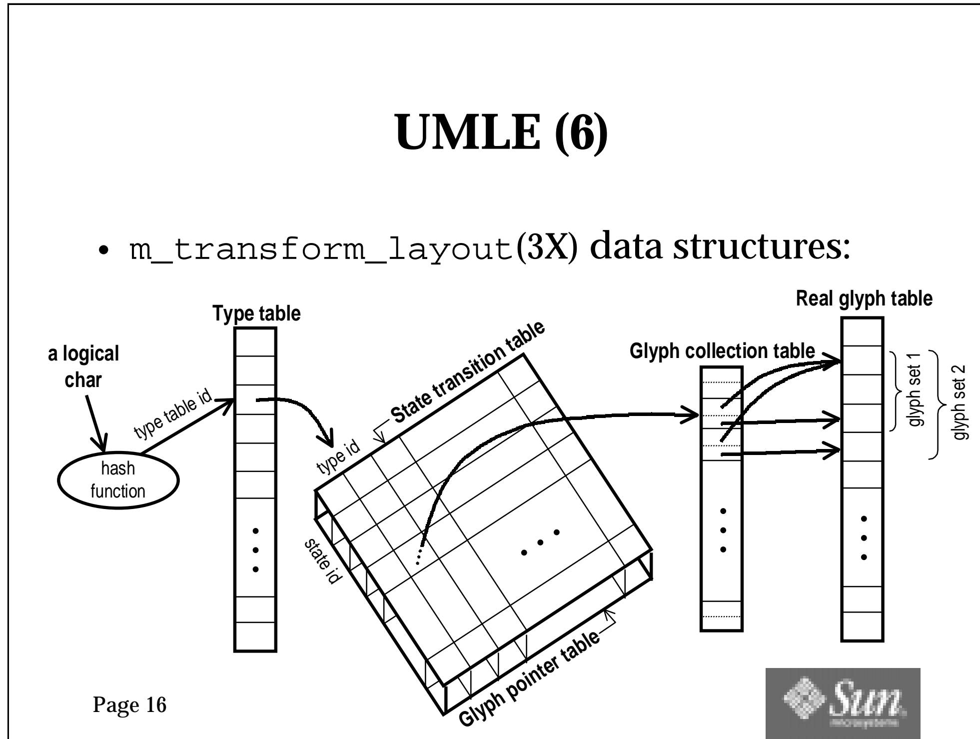
In the UMLE, `m_transform_layout`(3X) and `m_wtransform_layout`(3X) are the places where actual layout transformation occurs.

The `m_transform_layout`(3X) accepts file code based input text and the `m_wtransform_layout`(3X) accepts process code, i.e., wide character, based input text. Internal logic for these two functions are rather identical.

The transformation functions contain two basic subroutines/steps:

- *Reordering step* by using Unicode Bidi Algorithm.
- *Shaping step* by using Sun-proprietary table-driven shaping algorithm..

The goal of the transformation functions is to convert from one form of text to another form of output text for various purposes like actual rendering. The picture in this slide shows a possible case of such transformations.



Since the Unicode Bidi Algorithm is a well known one, in this and the next slides, we will only describe the Sun-proprietary, table-driven shaping algorithm and its key data structures.

Basically we believe every context-sensitive shape transformation is deterministic and finite in nature. We also believe that in such a transformation, by using a finite state machine, a series of input codes can be used to reach to a final state that points to a transformed output text, e.g., a transformed presentation form text consists of one or more of (font id, glyph id) pairs.

We defined following three key data structures for the finite state machine:

• Type table: A vector that has a type id for each code point of codeset. (To have space optimal data structure, we use dense encoding as the indices' encoding.)

• State transition table: A matrix with type id as column index and state id as row index. Each component of the matrix has a next state or a final state indicator as the value.

• Glyph tables: A matrix and several vectors that will provide a set of transformed output texts.

UMLE (7)

- `m_transform_layout`(3X) algorithm:

```
begin procedure table_driven_shaping()
        Sn <- Sprev <- S0
        loop
                Ch <- get_an_input_char()
                Ti <- type_function(Ch)
                Sn <- state_transition_table[Sprev <- Sn][Ti]
                if (Ch made Sprev a final state) then
                        unget_an_input_char(Ch)
                        break
                end
                push(Stack, Ch)
        until (Sn or Sprev is a final state)

        OutBuf <- collect_glyphs(Sn, Ti, Stack)
        return OutBuf
end
```

Page 17

Initial state of the finite state machine is S_0. Upon input of a code, we change to the next state until there is no more input or, a final state has reached by using a few lines of codes and functions in serial manner as shown in the `until` loop at the algorithm in this slide.

Once we reach to a final state, we reference the glyph pointer table to pickup an index for glyph collection table that contains begin index and end index of the real glyph table like following pseudo codes:

```
Id <- glyph_pointer_table[Sn][Ti]
StartId <- glyph_collection_table[Id].start
EndId <- glyph_collection_table[Id].end

while (StartId <= EndId)
        OutBuf[Id] <- real_glyph_table[StartId]
        Id <- Id + 1
        StartId <- StartId + 1
end
return OutBuf
```

By copying over specified amount of transformed output text codes, we achieve the desired transformation.

UMLE (8)

- `m_destroy_layout`(3X):
  - Release the allocated system resources back to the system.

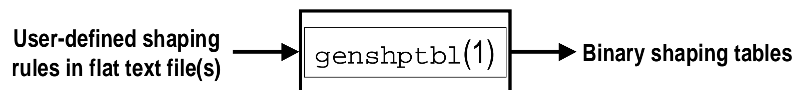
Page 18

By using m_destroy_layout(3) function, user can return and release allocated system resources back to the system.

For more detail on this and other layout services functions, refer to either corresponding system man pages or *X/Open CAE Specification -- Portable Layout Services: Context-dependent and Directional Text.*

UMLE (9)

- `genshptbl`(1):
  - A utility that generates user-defined, binary shaping tables for the UMLE.
  - User defines the shaping rules by using:
    - Character type definitions
    - State transition diagram definitions



Page 19

The `genshptbl`(1) is the utility that accept user-defined shaping rules and generates a set of binary shaping rule tables that can be loaded and used by the UMLE.

User defines the shaping rule by usually following steps at below:

• Classify each characters of the locale's codeset into a certain number of *types*. Characters in a type have a common property. For instance, Arabic characters can be categorized into several types like "4-shape type" characters can have four shapes (isolated, initial, middle, and, final forms), "2-shape type" characters can have two shapes (isolated and final forms), "1-shape type" characters can have only one single shape (isolated form or characters from other scripts), and so on.

• Enumerate *all possible state transition combinations* by using the types as input for each state transition, for instance:

S_0 -- Type1 input -> S_1 -- Type2 input -> S_2 … -> S_{final}

S_0 -- Type2 input -> S_2 -- Type3 input -> S_5 … -> S_{final}

:

• Once state transition combinations are done, define *all possible transformed output text* for each final states.

• Write above three definitions into a flat text file by following `genshptbl`(1) input file format defined in `genshptbl`(5) man page.

Demonstration

- Initial behavior of American English Unicode locale (en_US.UTF-8).
- Change of `Layout.conf` and shaping rules.
- Table generation by using `genshptbl`(1).
- Modified behavior of the American English Unicode locale.

Page 20

Comparison

	UMLE approach	Locale-specific layout engine approach
Easy to maintain	yes	no
Easy to add new CTL scripts	yes	no
User-defined CTL locale support	yes	difficult
Support of user customization	yes	difficult
Performance	quite acceptable	various (good to acceptable)

Page 21

Because the UMLE is specifically designed to achieve the three goals, codeset independent, multiscript capable, and, user-customizable, that are described in the slide page 11, it is much more flexible and easy to maintain. A few aspects of UMLE approach and locale-specific layout engine approach are compared in this slide.

Concluding Remarks

- UMLE is universal, i.e., *codeset independent*, *user-customizable*, and, *multiscript capable.*
- Any CTL scripts can be supported by the UMLE and `genshptbl`(1).
- Users can use the UMLE and `genshptbl`(1) to create or customize a CTL support in their locales.

Q&A

Page 23