Metadata and Interoperability Guidelines

Draft 1.10, 18 April, 2009
Originally Authored By:
Ken Ray, kray@sonsothunder.com
Richard Gaskin, ambassador@fourthworld.com
Contributions by:
Dick Kriesel, dick.kriesel@mail.com
Eric Chatonet, eric.chatonet@mac.com
Jan Schenkel, jan.schenkel@quartam.com

Maintained at:
RevInterop Group at Yahoo, http://groups.yahoo.com/group/revInterop

INTRODUCTION

As the Transcript development community grows we can expect many new innovative solutions coming from its members. These include:

Libraries
Stack files that hold scripts which can be called from throughout the message path, initialized with "start using".

Components
Objects, groups of objects, or stacks that have a specific purpose or function, such as a table widget or FTP settings dialog, useful across multiple projects.

Templates
Prefabricated collections of objects that work together to provide the basis for a complete software application, such as a shell for a slide show viewer or a system utility.

Collectively, these solutions are referred to as **resources**. As the variety of these things grows issues of interoperability are introduced, such as name space conflicts among handlers or objects. In addition to minimizing errors, there is an opportunity to provide easier integration of resources from different third parties.

This initiative proposes a set of conventions that aim to serve both goals, minimizing the risk of introducing errors when integrating resources, and exploring ways to make it simpler to use such resources.

Much of what is needed for that goal could be described generically as "metadata": additional information bound to the resources in question but which do not directly affect performance of those resources.

The Dublin Core Metadata Initiative (DCMI) is a consortium effort to facilitate exchange of documents. Given that the goals of the DCMI and this initiative are in many ways very similar, and that the DCMI working group has already expended tremendous investment in defining metadata needs for such purposes, this initiative aims to use DCMI nomenclature and conventions whenever practical.

To reinforce this deference to the DCMI project and as a tip-of-the-hat to the company that produces Transcript and Revolution (Runtime Revolution Ltd. of Edinburgh, Scotland), this project is given the lighthearted name of "Edinburgh Core Metadata Initiative" ("ECMI").

GOALS OVERVIEW - Must-Haves and Nice-to-Haves

Above all else programming with Transcript is simple, and nothing in this proposal should introduce requirements beyond anything required by the Revolution engine to execute.

It should be possible to completely ignore the recommendations of this proposal and still have acceptable performance of a library, component, or template, at least by itself.

Where the ECMI becomes useful is in anticipating areas of potential conflict between such resources, and anticipating that as the variety of third-party resources grows it will be desirable to have ever-simpler ways of working with them.

The recommendations of this proposal are therefore entirely optional, falling firmly into the Nice-to-Have category with ideally no Must-Haves.

That said, adopting the recommendations of this proposal may be seen as allowing a resource to be a "good citizen" within the larger host software it may be used in.

It's hoped that the ECMI proposal will reflect the needs of most, if not all, third-party Transcript programmers, as and such be widely used by them in works delivered to the community.

GOALS SPECIFICS

When sharing software resources it can be useful to have immediate access to a variety of information about the resource, including:

- Who made it, and how can I contact them?
- What does it do?
- How do I use it?
- What version is it?
- Where can I get the latest version?
- What are the legal terms of use?
- What are its technical requirements and dependencies?

Many of these are already defined in the DCMI, at <http://dublincore.org/documents/dcmi-terms/>. Since the DCMI is focused on documents rather than executable code, items like version and technical requirements must be defined here in the ECMI.

Since all objects in Revolution can have custom properties, it is proposed that custom properties be used for the storage of metadata.

To minimize potential conflicts with existing properties or any code that addresses them, it's proposed that ECMI properties be stored in their own custom property set.

In keeping with the long-established convention of prefacing user-defined (as opposed to engine-defined) properties with a lower-case "u", the ECMI custom property set is labeled "uRIP" ("RIP" for "Rev Interop"), and its elements can be accessed simply with array notation:

```
get the uRIP["version"] of this stack
```

For consistency, the uRIP custom property set should be included in the highest-level object for the software resource in question. For example, a component widget (like a "table" widget) would have the

uRIP custom property set in the group which contains its objects; a component stack (like an FTP Settings dialog), a library or template would have the uRIP custom property set in the mainstack of the stack file.

In addition to listing specific metadata elements, recommendations for useful naming conventions are provided to minimize potential conflicts and make software resources easily identifiable.

DEVELOPMENT OPPORTUNITIES

Once this specification has been officially released, there will be opportunities for developers to create add-on utilities related to ECMI. These can include:

- Tools to help in the creation and/or storage of ECMI metadata
- Tools to help in the creation of ECMI-compliant resources
- Tools to display ECMI data for a number of resources in one place
- Tools to locate and download ECMI-compliant resources

Although this is beyond the scope of this document, having the ECMI standard will allow for the development and distribution of these kinds of tools to aid developers in working with ECMI resources.

METADATA ELEMENTS

The metadata elements listed below form an initial proposed set useful for minimal interoperability. (Note: Those items with "-- DCMI" after their name are borrowed from the DCMI specification at <http://dublincore.org/documents/2004/06/14/dcmi-terms/>.)

Most of the properties are simple text strings, however any data that is wrapped in " " and " " suggest that the content of the property is to be interpreted as htmlText. It is therefore the responsibility of any tools that work with these properties to take this into account.

All of these are fully optional, and can be used when the developer feels it adds value to the resource being distributed. For the purposes of classification, however, these metadata elements have been grouped into "recommended" and "suggested" categories:

Recommended Elements:

`uRIP["name"]`
: The name of the resource. Included to have a single place to look to get the resource name, without needing to know if the resource is a component, template, or library.

`uRIP["copyright"] -- DCMI`
: Copyright statement.

`uRIP["creator"] -- DCMI`
: Name of individual or company responsible for creating the resource. Additional contact information can be provided, but line 1 must be the name of the individual/company.

`uRIP["date"] -- DCMI`
: Release date of the resource. Date is in the format YYYY,MM,DD (the first three items returned from the dateItems format); padding with zeroes for MM or DD < 10 is optional.

`uRIP["description"] -- DCMI`
: Summary description of what the resource does. For our uses probably best to use a simple, single-sentence description similar to what one finds in the second line of an About box.

`uRIP["help"]`
: Instructions on how to use the resource.

`uRIP["homeURL"]`
: Uniform resource locator to the WWW home page providing information about the resource.

`uRIP["downloadURL"]`

Uniform resource locator to directly download the latest edition of the resource. This is different from the Home resource, which points to the web page describing the resource.

`uRIP["interface"]`

Human-readable return-delimited list of handlers and functions with descriptive parameter names that can be called from other software resources. It is suggested that each line in the list follow the standard syntax structure:

- Variables are enclosed in angle brackets <>
- Optional parameters are enclosed in square brackets []
- Parameters that must be one of a list of items are in curly brackets (braces) {} separated by vertical bars |
- Parameters should have descriptive names
- Parameters that are generic object descriptors use "objectDescriptor", "objectDesc", or "objDesc"
- Parameters that are specific object descriptors use a similar form, but specify the object type, as in "fieldDescriptor", "buttonDesc", or "grcDesc"
- Functions should put all params inside parentheses

Examples:

```
stsImg_ConvertPolygon <objDesc>,{"bezier"|"quadratic"}
stsImg_GetHandles(<objDesc>)
```

`uRIP["EULA"]`

A legal document ("End User License Agreement") giving official permission to do something with the resource.

`uRIP["requirements"]`

List of system requirements and dependencies, where each line is in the form:

```
{"stack"|"library"|"external"|"file"},<nameOrPath>
```

where:

`stack` is a requirement that a certain stack already be open (i.e. in "the openStacks")

`library` is a requirement that a certain library already be in use (i.e. in "the libraryStacks")

`external` is a requrement that a certain external is loaded (i.e. is in "the externals")

`file` is a requirement that a certain file needs to exist at a particular location (i.e. "there is a file <path>" returns "true")

<nameOrPath> is either the name of the required item (like for stacks, the name of a stack), or is a path to the required item (for example, DLLs or externals).

Examples:

```
stack,Options
library,libSTSGeneral
external,revXML.dll
file,../../test.doc
```

`uRIP["version"]`

The version number of the resource, preferably in x.x.x notation.

`uRIP["buildNumber"]`

The build number of the resource, preferably as an integer.

`uRIP["specVersion"]`
The version number of the RIP Specification to which the resource conforms.

Suggested Elements:

`uRIP["type"]`
The custom "type" of the resource. This is most often used with components (as opposed to templates or libraries). This can be any method of identifying the resource, but it is suggested that it be one or two words, and include a similar naming convention to that of handlers in libraries (see "Naming Components" in the "Naming Conventions" section below).

`uRIP["updateInfoURL"]`
Uniform resource locator to a plain text file on a web server that contains information related to the current version of the resource. The format of the file is XML[1] in this format:

```
<rip>
   <file>
      <name>resourceName</name>
      <version>currentVersionNumber</version>
      <buildNumber>currentBuildNumber</buildNumber>
      <releaseDate>releaseDate</releaseDate>
      <updaterURL>updaterURL</updaterURL>
      <downloadURL>downloadURL</downloadURL>
      <releaseNotes>releaseNotes</releaseNotes>
   </file>
</rip>
```

where:

name is the name of the resource.

currentVersionNumber is the version number of the current version of the resource. Usually this is in x.x.x notation.

currentBuildNumber is the build number of the current version of the resource. Usually is an integer value.

releaseDate is the date (or date and time) that the current version of the resource was released. The date is in the format YYYY,MM,DD (the first three items returned from the dateItems format); padding with zeroes for MM or DD < 10 is optional.

updaterURL is the full URL to a Revolution stack that will be displayed if it is determined that an update to the resource is necessary.

downloadURL is the full URL used to download the current version of the resource (e.g. a stack file, a compressed archive, an installer executable or .dmg, etc.). This can also optionally contain a URL reference to a web page where the current version of the resource can be downloaded.

releaseNotes is a list of things that changed between the previous release of the resource and the current release. This can also optionally contain a URL reference to a web page that will display the release notes; and should be full URL form starting with "http://". Also note that if you have multiple lines in the *releaseNotes*, you will need to wrap the information in

[1] *Note that even though the contents of the file is XML, the file extension is ".txt". The reason for this is that many web servers will not serve a ".xml" file when you use a simple Revolution "get url" call (you generally get an Error 500). So to avoid this, the file has a ".txt" extension.*

CDATA tags so that the data will not be normalized and "squished" together (see http://w3schools.com/xml/xml_cdata.asp).

Note that as you can have multiple `<file></file>` trees under the `<rip>` node if you want to have a single update file handle multiple resources, or if you want to have a "version history" update file that handles multiple versions of the same resource.

Example:

```
<rip>
   <file>
      <name>MyTool</name>
      <version>1.0</version>
      <buildNumber></buildNumber>
      <releaseDate>2005,6,21</releaseDate>
      <updaterURL>http://www.fourthworld.net/updater.rev</updaterURL>
      <downloadURL>http://www.fourthworld.net/mytool.rev</downloadURL>
      <releaseNotes>- Fixed startup crashing bug.</releaseNotes>
   </file>
   <file>
      <name>MyOtherTool</name>
      <version>1.5</version>
      <buildNumber>23</buildNumber>
      <releaseDate>2005,6,20</releaseDate>
      <updaterURL></updaterURL>
      <downloadURL>http://www.fourthworld.net/other.htm</downloadURL>
      <releaseNotes>
         <![CDATA[- Changed the term "Widget" to "Thingy."
         - Fixed startup crashing bug.
         - Applied universal fix for the "Make Application" button.]]>
      </releaseNotes>
   </file>
   <file>
      <name>MyOtherOtherTool</name>
      <version>1.2</version>
      <buildNumber>127</buildNumber>
      <releaseDate>2005,6,19</releaseDate>
      <updaterURL></updaterURL>
      <downloadURL>http://www.fourthworld.net/other2.rev</downloadURL>
      <releaseNotes>http://www.fourthworld.net/rel.htm</releaseNotes>
   </file>
</rip>
```

`uRIP["props"]`

A list of custom properties and definitions that defines the scope and restrictions for assigning these properties.

This metadata item is in place to allow third-party "property pickers" to allow the end user to select only from the valid set of values for a specific custom property. For example, if a component had a custom property of "priority", and the only valid values were "high", "medium", and "low", this definition would allow the property picker to display this as a drop-down menu with only these selections.

If this metadata item does not exist, the property picker would display a standard generic text editing field to allow the user to change the value of the component's custom property.

This metadata item may also be used by "strip-and-ship" or "reset" handlers to restore a resource to its factory settings if the `<propDefault>` parameter is provided (see below).

The value for this metadata item is return-delimited, where each line corresponds to a specific custom property and is in the form:

```
<propName>,<propLabel>,<propDefinition>[,<propDefault>[,<propDesc>]]
```

where:

<propName> is the name of the custom property, either in a custom property set or standing alone. The name should follow the naming conventions for custom properties (see "Naming Custom Properties" in the "Naming Conventions" section below).

<propLabel> is the text that should be shown in the property picker for the custom property.

<propDefinition> is the definition of what is supported for the custom property. This can be one of the following (symbols that should be typed as part of the definition are in bold; variables are in italics):

{*value1***, ... ,** *valueN***}**
: This represents a limited set of selectable values.
Example: {apple,orange,cherry}

`{number|integer}[:{positive|negative|any|range:{`*minNum***+**`|`*maxNum***-**`|`*startNum* `to` *endNum*`}}]`
: This means that the value for the custom property must be either a number (real) or integer, and whether it must be positive, negative, any, or a range. Ranges can either be from a minimum number upward, and maximum number downward, or as a range between one number and another. If used as just "number" or "integer", any number or integer value will be accepted.
Examples: number
integer:positive
integer:range:3 to 10
number:range:0+
integer:range:3-

`text`
: This means that the value for the custom property can be a string of any length, even multiple lines.

`color`
: This means that a color picker should be displayed to set the value of this custom property.

`font`
: This means that a list of available fonts should be displayed to set the value of this custom property.

`trueFalse`
: This means that a checkbox could be used to set the value of this custom property. (Possible alternative definition names: "TF", "boolean", "bool" ?)

`rect`
: This means that the value for the custom property must be 4 numbers separated by commas.

<*expression***>**
: This means that the value for the custom property will come from a list that is generated by evalutating the expression <expression>. This can be something like "the cardNames", or "the properties of card button 1". This must be something that "do" can "put" in the Transcript statement:

```
do "put" && tExpression && "into tListToDisplay"
```

[*handler***]**

This means that the value for the custom property will come from a list that is generated by calling the handler <handler> in the resource. This would most commonly be used for handling property values that are too complex to be supported by the other property definitions.

The script that this handler resides in will be the same place where the uRIP custom property set is found; for example, libraries and templates would have this handler in the stack script, and components would usually have this handler in the group's script (if implemented as a group), or in the object's script (if implemented as an object).

Note that the handler must "return" the value so that the ECMI-compliant property picker can display the result properly.

<propDefault> is the default value for the property. This is optional, and can be used by "strip-n-ship" or "reset" handlers to restore the resource to its default "factory settings". This is especially important for applications where non-standalone stacks are saved to disk; it provides the ability to override properties that had been saved with incorrect values. Note: If the default value contains a comma, it should be quoted.

<propDesc> is an optional description for the property that might be displayed in a custom property picker interface; this would be the equivalent of a "tooltip" and provides a place for a more extensive description than the <propLabel> would allow. Note: If the description contains a comma, it should be quoted.

Examples:

```
uSTSPriority,Priority,{high,med,low},med
uSTSActive,Active,trueFalse,true
uSTSAge,Age,{10-100}
uSTSColor,Color,color
uSTSWindowRect,Window Rect,rect,"100,100,300,200"
uSTSTargetCard,Target Card,<the cardNames>
uSTSActiveUser,Active User,[GetActiveUsers],,The name of the user
   currently logged into the database.
```

NAMING CONVENTIONS

In the conventions listed below there are a few placeholders that are used. They are:

`<company>`

This is a 2-5 character designator or abbreviation, in all capitals or initial capitals, that indicates the company or entity that has produced the resource. For example, *Sons of Thunder Software* might use "STS", and *Fourth World* might use "4W" or "FW". (It is recommended that you use at least 3 characters to allow for more diversity in what prefixes are "booked".)

`<lowerCompany>`

This is the same as <company>, but is in all lower case.

`<fullResName>`

This is the full name of the resource, and is usually one or two words with no spaces and initial capitals. For example, a library related to image manipulation might be called "ImageManager"; a table component might be called "Table"; a template for a Search dialog might be called "SearchWindow".

<abbrResName>

This is an abbreviated name of the resource, and is usually 2-5 characters in initial capitals. For example, a library whose <fullResName> is "ImageManager" might have an <abbrResName> of "Img"; a table component called "Table" might have an <abbrResName> of "Table" or "Tbl"; a template for a search dialog whose <fullResName> is "SearchWindow" might have an <abbrResName> of "Srch".

Naming Libraries

All libraries should be named "lib<company><fullResName>" (an acceptable alternative is to put an underscore (_) between <company> and <fullResName>). Examples are "libSTSImageManager" or "lib4W_Files".

Naming Handlers in Libraries

There are two kinds of handlers in libraries - those functions or handlers that are private (i.e. should only be used by other "public" functions/handlers in the library and should not be called on by any other stack), and those that are public.

Public handlers should be named <lowerCompany><abbrResName>_<handlername>. For example, a handler for the library "libSTSImageManager" that converted a polygon to a bezier curve might be called "stsImg_ConvertPolygon".

Private handlers are named similarly to public ones, but are preceded with an underscore (_). So to run a transformation on a series of polygon points and return a different set of points, the function might be called "_stsImg_TransformPoints".

Naming Custom Properties

Like handlers in libraries, there are those custom properties that are public (i.e. intended on being set by the developer), and those that are private (those intended for storage but not manipulation by the developer).

Public custom properties for libraries, templates, components, etc. are in the format:

"u"<company><abbrResName>_<PropertyName>

As an alternative, you can use a single custom property set, with individual custom keys related to the custom properties you wish to set/get. This would be in the format:

"u"<company><abbrResName>["<propertyName>"]

(NOTE: It is strongly suggested that you use one or the other, but not both.)

So for example, a custom property identifying the number of columns for a table component provided by Sons of Thunder Software might be "uSTSTable_NumCols" (or uSTSTable["numCols"]). Another example is a custom property specifying the number of uses of a stack template by Fourth World might be "u4WTmpl_NumUses" (or u4WTmpl["numUses"]).

Private custom properties are named similarly to public ones, but are preceded with an underscore (_). So to store some data for a status bar component might have a custom property called _uSTSStatusBar_Data (or _uSTSStatusBar["data"]).

Note that private custom properties are rarely used, as there are generally better means of storing private data.

Naming Components

Components are either single controls (fields, buttons, etc.) or are groups of controls. When naming components, keep in mind that the end user will rename the component to fit within the context of their deployed application. As such, it is suggested that the component be named using the format <lowerCompany><fullResName>, so a table object component (which is a group of controls) released by *Sons of Thunder Software* might have the main group object named "stsTable".

You may wish to also adopt a popular naming method whereby the first component of this type on a card is named <name>1, the second is named <name>2, etc. So if there were two table object components by *Sons of Thunder Software*, the first would be named "stsTable1" and the second "stsTable2". It is easy enough to create code that will do that for you:

```
repeat with x = 1 to 1000  -- or some other high value
  if there is a group ("stsTable" & x) then next repeat
  set the name of last group to ("stsTable" & x)
  exit repeat
end repeat
```

It is further recommended that if you are implementing the uRIP["type"] property, that you set it to be <lowerCompany><fullResName> (in the example above, that would be "stsTable") so that regardless of whatever the user renames it to, you can always know what "type" of component it is.

Naming Controls in Components (for those that are groups)

For controls that are *inside* a component (for components that are released as group objects), it is suggested that they be named starting with an underscore (_). The reason for this is that any code that needs to work on a list of controls on the current card (for example something that hides all objects on the card), can skip over the "internal" controls by simply checking if the first character of the "short name" of the control begins with "_". For example, if the component "stsTable" were made up of two fields, a "header" field and a "body" field that are grouped together and operate together, the header field might be named "_Header", and the body field might be named "_Body".

If you are implementing the uRIP["type"] property for the component, you may wish to implement the same property for controls *inside* the component if you need to identify them. This is especially important when there are more than one of the same component on the card.

To do this, it is suggested that you set the value of the uRIP["type"] property of the internal object to <lowerCompany><fullResName>_<identifier>. So in the case of the stsTable control, the header object would have a uRIP["type"] of "stsTable_Header" and the body object would have a uRIP["type"] of "stsTable_Body".

Naming Templates

Templates have the least possibility of negative interactions with other templates, since by definition they are renamed as soon as they are implemented. Because of this, there are no real naming conventions for templates, other than suggesting that it include the word "template" or an abbreviated form of "template" somewhere in its name, and possibly a nod to the <company> that created it. For example, a template of a system utility might be called "stsSysUtilTemplate", or "4WSystemUtilityTMPL", etc.

STRUCTURAL CONVENTIONS

This section provides some guidelines to developing the structure of a resource. By "structure", it means the items that make up the resource, and suggestions on how they should be constructed and laid out in an interface (if applicable).

Libraries

When implementing a library, remember that when the library is put into use with "start using", the library stack's stack script is accessible, but the library stack itself is *not opened*; for that reason, there is technically no visible interface to a library when it is being used by an application.

However, you can also *open* a library stack (like any other stack) in the IDE during development. It is at this point that there should be some user interface to understanding the library and how it is used.

Lib4WShell, which is available in the Files area of the RevInterop group on Yahoo (http://groups.yahoo.com/group/revInterop/files), provides an excellent foundation for the structure of a library, and should be looked at for an initial approach to developing libraries. It consists of a mainstack which displays information about the library itself, and a substack for "notes" that is used for information on what the library is and how it is to be used.

Components

TBD

Templates

TBD

SUMMARY OF CHANGES SINCE LAST (1.01) DRAFT

- uRIP["updateInfoURL"]
    - Added `<buildNumber>` and `<updaterURL>`.
    - Added suggestion for possible use of multiple `<file></file>` trees to track version history for a single resource.
    - Changed the description of *downloadURL* to indicate what kind of things can be downloaded and that it can optionally point to a download page.
    - Added clarification of ".txt" vs. ".xml" (including footnote).
- uRIP["props"]
    - Added paragraph providing optional use of `<propDefault>` for strip-and-ship purposes.
    - Changed `<propName>` to include ability to use a custom property set and directions to use naming conventions.
    - Added optional `<propDefault>` and `<propDesc>` options, and expanded the "Examples" to include some examples of their use.
- Naming Conventions: Naming Custom Properties
    - Added support for using a custom property set instead of individual custom properties.