Tivoli

Global Verification Testing in Unicode and Non-Unicode Environments

David B. Kumhyr

Contents

1. In the Beginning

Since 1995 Tivoli Systems has been enabling and translating its software, documentation, educational and marketing materials. The first translation and enablement effort was a performed by a small team of engineers enabling and translating a version of Tivoli's key product suite resulting in TME 3.1 for Japanese. The resulting code changes were then merged into the main code source base.

Testing TME 3.1J was relatively simple since the team of engineers whose specialty was globalization did all of the testing. They were thoroughly familiar with enablement issues, the Japanese operating systems as well as the software they were testing.

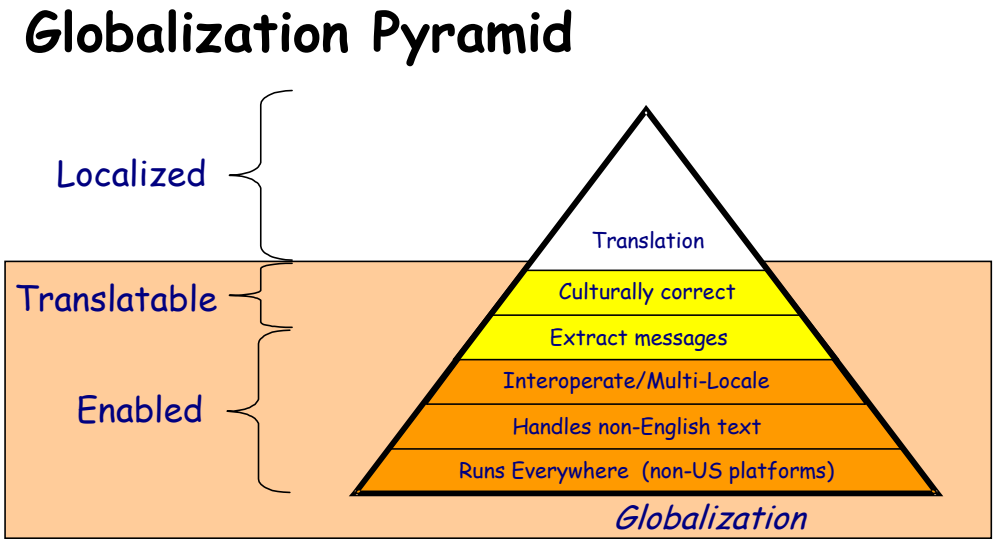
Since that time all of Tivoli's products are enabled to run anywhere and are translated into ten languages. The key enabling skills are still concentrated in a small team of specialists whose task it is to assist product development engineers enable their own code.

Besides the challenge of architecting a standard globalization model, the globalization team teaches development engineers to enable the base code and also must educate the verification teams to test for problems that may occur in foreign language environments. As a result the globalization team has developed a body of tools and techniques to aid in international testing.

This paper outlines the evolution of testing and the technology developed to cope as well as highlights their key features.

2. Creating an International Architecture

To meet the challenge of worldwide applications the globalization team developed enablement model and globalization architecture to unify and simplify multi-language development.



Our basic model is the Globalization Pyramid, which describes the hierarchical architecture. Tivoli's development globalization baseline requirements are derived from the pyramid.

Levels of the Globalization Pyramid

Customers want applications that run anywhere, meaning that they must be usable on any language operating system with non-US keyboards or other country-specific hardware. Applications should not have hard-coded dependencies on human language strings, and must inter-operate with non-US versions of other products. Applications should be codeset independent, be able to handle multi-byte characters and handle differences in a distributed environment.

Our applications support global enterprises by allowing multiple nodes executing in different languages and allowing multiple languages in persistent data repositories while transmitting and manipulating data in a culturally neutral manner. They also must permit local views to data according to viewer's language and cultural preferences, supporting multilingual text in graphical user interfaces.

To be easily translatable applications should isolated human language text into separate translatable files of standard formats. Icons and images are language-independent and culture-neutral. Text strings are easily translatable, message sentences are not assembled

from parts at run-time and the order of run-time variable references are flexible. Text is displayed using translation, which most closely matches viewer's language preference.

Information is presented to the user in a manner consistent with the user's local cultural conventions including data formatting, layouts, collation and directionality of text and graphical user interface (GUI) components.

Another major change to the original development style was the de-coupling of program code from human language by separating language into separately installable Language Packs. In this manner the impact of localization activities on base development is minimized. This technique also allows the entire localization process to be done by a separate organization or at a different location. Additionally new translations can be released at any time, independent of base product cycles.

3. Software Testing Phases

Tivoli's original English only development process included the following testing phases. Responsibility for performing these tests was shared between the development organisation and a special testing (verification) organisation.

Original Test Phases	**Basic Test Description**
Unit Test	Performed by the developer to verify the correct functionality of a software modules prior to checking it into the source code control system.
Build Verification Testing	Executed by either the development or build team on an official build before it is passed to verification for System Verification Test.
Functional Verification Testing	Verifies that the production build has working functional content and is ready to invest the resources to start System Testing. Components may arrive in FVT at different times and be tested separately.
System Verification Testing	Testing with all components of the products installed and completely integrated. Focus on discovering defects caused by interactions between components.
Performance Analysis and Bench marking	Performance testing attempts to discover performance problems in the product. Test cases perform stress and load testing. Performance bottlenecks are identified using specialised tools.
Integration Testing	Test the behaviour of multiple applications running concurrently.

When Tivoli began translating its products international testing became necessary which lead to adding the following test phases. This transition was not immediate but accomplished through an evolutionary process.

New Test Phase	**Basic Test Description**
Global Verification Testing	The portion of the FVT that addresses globalization issues. Assures the product can run in non-US environments and is ready for world-wide distribution by the ESP. GVT should finish before start of TVT so severe or blocking defects can be fixed, tested and built.
Translation Verification Testing	The linguistic verification of the translation of a product.

4. The Evolution of International Testing

Linguistic Testing

With translation came the need to verify the translated versions. Translation Verification Testing (TVT) was introduced. It was modelled on the process of linguistic verification done at IBM.

Currently TVT at Tivoli is a cumbersome and expensive operation involving bringing translators from their countries to the development site. The product under test must be installed and configured and test cases created with sufficient documentation for translators unfamiliar with the operation if the product to manually verify in context all of the translated text. Not only does TVT involve enormous expense in hardware, travel and personnel the timing of the test is a severe burden on the development and testing organisations. Hourly charges for test run into the thousands of dollars.

Due the inexperience of the development teams with enablement, during the conduct of TVT functional defects are often discovered. Since TVT occurs after development has frozen the code and following translation finding defects this late makes them difficult and expensive to fix. If the defect is severe enough it may temporarily halt the TVT testing, further raising the costs by idling translators and testers and extending the dates for the test.

Functional Testing

To help to prevent TVT cost overruns and delays some type of functional testing with an international focus had to be instituted in the development testing phases. Globalization Verification Testing (GVT[1]) was created and mandated during the Functional Verification Testing (FVT) phase of development.

Development testers, with no previous experience with international function testing need assistance to complete these tests. The globalization team stepped in creating the specifications, a base test suite and tools with which to perform the GVT task.

The goal of GVT is to identify as early as possible functional defects related to international operation. A GVT is a portion of the product functional verification test (FVT) that addresses globalization issues. This test assures the product can run in non-US environments and after translation. Since it would be a near impossibility to verify all language versions of all operating systems it is not practical to run a full GVT of a product in all environments. GVT testing incorporates the use of specialized tools to verify functions in different environments.

[1] Some Tuesday afternoons my team has taken GVT to mean Golf course Verification Test.

5. International Testing Tools

Obviously early detection and correction of defects saves time, money and customer goodwill. Early intervention is especially vital if the problems are architectural or endemic. Accordingly we have focussed our primary efforts on early detection of enablement problems.

Each development project has an globalization engineer assigned as early as possible[2] to work with the product development team. However due to the limited number of globalization engineers and the large number of products globalization engineers work on multiple projects and the amount of time the engineer can spend with each team is limited.

During early development the globalization engineer can review internal web-based checklists and documents covering enablement and translation with development team members. This gives the teams a clear idea of the rules that must be followed for successful enablement.

In addition the globalization engineer will scan the source code of the product with a tool that identifies potential enablement problems. Tivoli uses the OneRealm[3] I18N Expeditor™ and I18n Reporter™ tools for this purpose. The reports from these tools are voluminous and need to be reviewed by someone skilled in their use. The globalization engineer works with developers from the product team who knows the product source code to identify the real problems from the reports. The development team then fixes these problems with assistance of the globalization engineer.

Two of the testing tools we employ are based upon the concept of running the application on a non-US operating environment and using a mock translation. The first using verification through execution tests that tests the applications enablement by running on a selected foreign language operating system. Its purpose is to verify specific functional support such as bi-directional language support, Unicode or multi-byte character set support.

Secondly, since actual translations are not ready until late in the development cycle we have employed mock translation to approximate a translation. Our first mock translation tool involved a simple parser that transformed the English language text of the application being tested by padding with additional characters. First a delimiter character '[' is added to the beginning and end of the string. This allows the tester to check for truncation or composing of messages. Next the string is padded according to an expansion algorithm with up to 200% of a multi-byte '~' character that has as one byte a hexadecimal 'C5' character. This expansion allows testing for truncation of messages and expose byte by byte parsing of strings. Finally vowels are changed to an accented version.

Sample Dialogue Box - Untranslated

[2] Usually when the design documents are first written, though later with projects added through acquisition.
[3] OneRealm Inc, Boulder Colorado, USA

```
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 235, 55
STYLE DS_MODALFRAME | DS_CONTEXTHELP | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "About IBM Calendar Support 1.0"
FONT 8, "MS Sans Serif"
BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,20,20
    LTEXT           "IBM Calendar Support 1.0",IDC_STATIC,40,10,119,8,
                    SS_NOPREFIX
    LTEXT           "Copyright (C) 2000, IBM",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "OK",IDOK,178,7,50,14,WS_GROUP
END
```

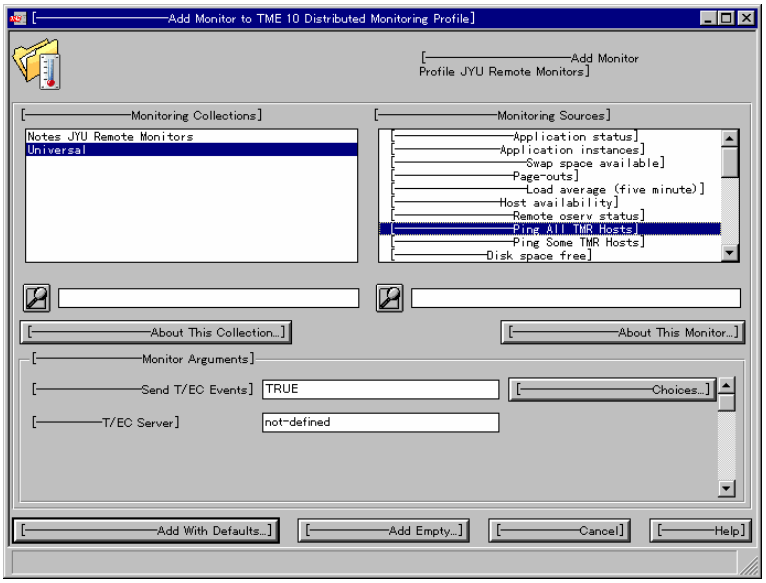
Sample Dialogue Box - Mock Translated

```
IDD_ABOUTBOX DIALOG DISCARDABLE  0, 0, 235, 55
STYLE DS_MODALFRAME | DS_CONTEXTHELP | WS_POPUP | WS_CAPTION | WS_SYSMENU
CAPTION "[~~~~~~~~~~~~~~~~~~~~~~~~~Ab\xf6\xfct IBM C\xe4lend\xe4r S\xfcpp\xf6rt
1.0]"

BEGIN
    ICON            IDR_MAINFRAME,IDC_STATIC,11,17,20,20
    LTEXT           "[~~~~~~~~~~~~~~~~~~~IBM C\xe4lend\xe4r S\xfcpp\xf6rt
1.0]",IDC_STATIC,40,10,119,8,
                    SS_NOPREFIX
    LTEXT           "[~~~~~~~~~~~~~~~~~~Copyright (C) 2000,
IBM]",IDC_STATIC,40,25,119,8
    DEFPUSHBUTTON   "[~~~~OK]",IDOK,178,7,50,14,WS_GROUP
END
```

This technology will catch the enablement defects of hardcoded strings, composed messages as well as missing messages catalogues or language files. In a GUI it will expose expansion and alignment errors and truncation problems. However it will not catch all enablement defects.

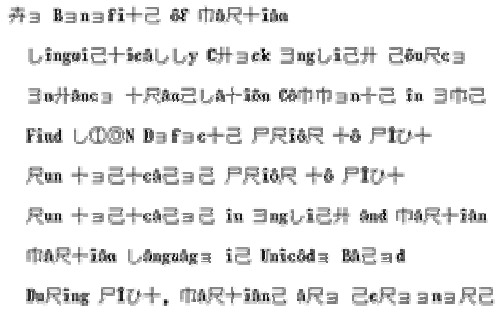
Below is a sample dialogue with mock-translated text.



A major failing of our early mock translation was that it didn't expose some of the key functional failings in enablement. The majority being related to locale specific behavior, such as data formatting (date, time, numeric), sorting and collation and some functional problems.

Discussions about how to increase the efficiency of GVT testing tools lead to wider research across groups within our IBM companies. The IBM AIX group is Austin Texas contributed a mock testing locale and language called Martian. It's implementation was limited to AIX and applications using ZPG4 message catalogues for storage of human language text.

Our globalization team expanded the locale support and created parsing tools to run on Windows and Java. and have used it successfully in the development of several Tivoli products to test enabling. Martian has a character set that is composed of 'look alike' characters that are abstract approximations of the shapes of our Roman alphabet using a combination of accented, combining and Asian characters.[4]



[4] An interesting side effect of the language is that while English speaking Westerners can read the language it is extremely difficult for Asians who read Chinese or Japanese to read. Since the characters have actual intrinsic meaning it requires much concentration to dismiss the meaning and read it as stylised English text.

To test using Martian the applications textual information is *translated*[5] by passing it through a parser that converts the information to Martian. The Martian parser converts non-programmatic information in Windows resource files, Java resource bundles, XPG4 message catalogues, HTML and XML files into Martian Unicode characters.

Sample Source Java File

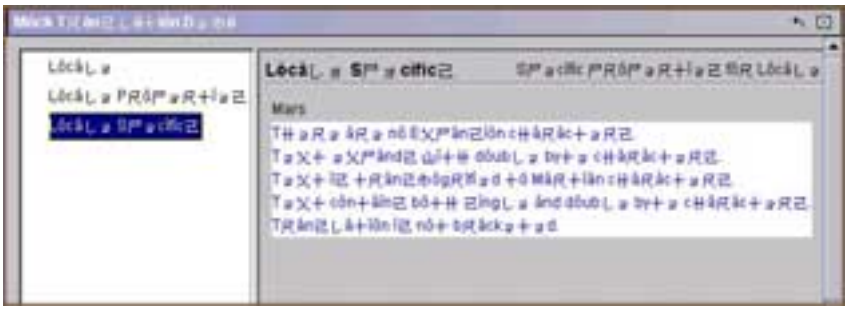
```
private static final Object[][] contents_ =  {
        // LOCALIZE THIS
        { NLS_DISPLAY_NAME, "Mock Translation Demo" },
        { NLS_SHORT_DESC, "Mock Translation Demo Properties" },
        { NLS_MOCKLOC_GENERAL_SHEET_LABEL, "Locale" },
```

Sample Martian Java File

```
private static final Object[][] contents_ =  {
        // LOCALIZE THIS
        { NLS_DISPLAY_NAME, "M\u00f4ck
T\u5c3a\u00e2n\u5df1\u3057\u00e2\u5341\u00ee\u00f4n D\u30e7\u5dfe\u00f4" },
        { NLS_SHORT_DESC, "M\u00f4ck
T\u5c3a\u00e2n\u5df1\u3057\u00e2\u5341\u00ee\u00f4n D\u30e7\u5dfe\u00f4
P\u5c3a\u00f4\u5c38\u30e7\u5c3a\u5341\u00ee\u30e7\u5df1" },
        { NLS_MOCKLOC_GENERAL_SHEET_LABEL, "L\u00f4c\u00e2\u3057\u30e7" },
```

The output of the *translation* is checked into source code control, then extracted and submitted as a Language Pack build with the locale of ma_RP[6]. The output of a successful build will be an installable Language Pack that can be applied and tested.

Once applied the application should display the translated interface, any portion of the interface that does not display Martian is either hard coded or not honouring the locale change. Dates, numeric data and collation sequences are different for the Red Planet locale to enable the tester to identify locale-honouring problems also. The figure below illustrates a Martian translated application that displays hard coded English text "Mars" in its interface.



[5] Translated exactly as the source will be translated during the actual translation process.

[6] The language is Martian and the locale is Red Planet.

6. The Next Steps in Test Evolution

To increase efficiency and reduce errors in translation and enablement we have three strategies; education of developers, increased efficiency in discovering errors as early as possible and reducing costs of finding translation verification errors.

Education

Through education our team can increase awareness of enablement and it's importance to Tivoli's strategic direction. Tivoli's executives assist us in reinforcing the need to enable and translate by the reporting of the companies quarterly results divided by geographic market area. This makes it obvious to everyone in the company that a great deal of our income is from worldwide sources.

Our team conducts classes in globalization and enablement on both formal and informal schedules. Formal classes are scheduled and announced at each Tivoli site during the year, often paired with IBM internal education classes on National Language Support enablement. Informal classes are often arranged with the development teams as requested. These are tailored to their individual needs. However turnover of developers and the lack of any enablement education in computer science education is a hindrance.

Additionally all of our enablement processes are peer reviewed, fully documented and kept under change control. Access is available through Tivoli's intranet. This rapidly available controlled information assists developers in getting started with projects with the correct enablement architecture.

Efficiencies in Discovering Errors

Further enhancement of the testing tools and development of standard test templates will assist in better verification and more thorough international testing. In the future we look at separate international test phases becoming merged into the normal verification that is done in the normal course of work rather than being a separate process.

Reducing Costs for Translation Verification

The most time consuming and expensive part of the translation process is Translation Verification Testing, therefore the greatest efficiencies can be gained in this phase. The largest gain will be made through cutting costs in linguistic verification, with the goal of eliminating TVT totally. A number of strategies are being researched.

First the outsourcing of TVT to a subcontract vendor. The major difficulty is finding a vendor with the translator and hardware pool large enough for the projects as well as a level of technical expertise to handle complex systems management software projects.

If one could eliminate the need to run the software on the actual platform and rather simulate the platform so that all or most of the human language text could be verified in context TVT would not have to face the large hurdles of hardware and expertise and instead concentrate on simply verifying text the savings would be great.

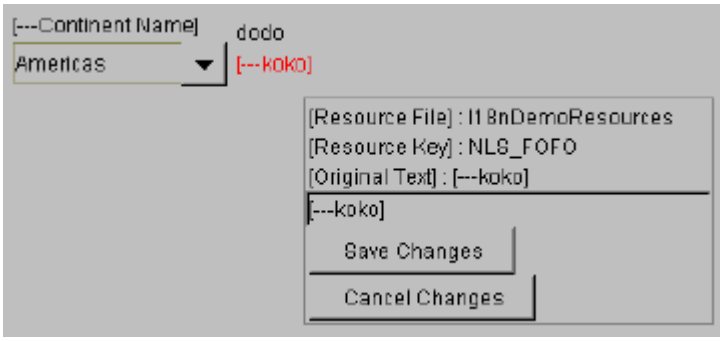
The two most promising technical strategies to accomplish this are journalling and a discovery that we are researching named Introspective Editing[7].

Journalling

Journalling involves saving and playing back mouse movements and keystrokes[8]. An experienced tester can execute a number of test suites on the original version of the software and the recorded journal can be played back against the translated versions. The translator doesn't need expertise with the program to view the human language text for verification. However this scheme still requires that the full system hardware and resources be available so execution can complete without resource errors.

Introspective Editing

Introspective Editor which uses the Java feature of introspection to 'reach inside' the Java based program and allow a translator to edit the applications interface on screen and then save and view the changes. Additionally the Introspective Editor presents the possibilities of doing TVT testing in country since the introspection enables the application to be installed and edited without a major test bed investment. This eliminates the need to actually run the program to view the interface, search for the text and edit the correct file. This reduces the need for hardware, knowledge of the program and saves errors and much time.



Current problems that we are working to solve are the building and populating of dialogues that happens during application run time. Combining the techniques of Introspective Editing with Journalling holds out the greatest hope of solving this difficult problem.

[7] Multiple patent submissions and research work by David Kumhyr, Keiichi Yamamoto, Dr. Jim Yu & Stanford Yates, alias Y[3]K.

[8] The data is captured by message on the message queue rather than by the location or character.

7. Appendix

Acknowledgements:

I would like to gratefully acknowledge the assistance of many people whose efforts have led to developing the processes and techniques described in this paper. They are not listed in order of importance.

Frank Rojas, Manager Tivoli Globalization Services
The Tivoli Internationalization and Tools Team (Dae-Suk, Greg, Bob, Hosheng, Mary, Tony, Steve, Lee, Keiichi, Dalal and Jim)
IBM Advanced Workstations and Systems Group
Y3K Inventors (Jim Yu, Keiichi Yamamoto, Stanford Yates et. Al)

For more information:

David B. Kumhyr
david.kumhyr@tivoli.com
Telephone +1 512/436-1268

Definitions:

Internationalized (Enabled)
- Development of products that are independent of language, culture and regional text encoding.
- Use globalization services to access appropriate localization features at run-time.
- Responsibility of base product development.

Localized - subset of languages dependent upon market.
- Development of localization features which can be installed on top of base product to provide correct local behavior.
- Responsibility of localization teams.

Language Pack
- Individually installable language versions.
- Separately buildable from product.