A Framework for Global Software [*]

Aarno Lehtola, and Timo Honkela[**]

VTT Information Technology
P.O.Box 1201, FIN - 02044 VTT, Finland
Tel: +358-0-4566032. Fax: +358-0-4566027
Email: aarno.lehtola@vtt.fi

This presentation outlines a *Framework for Global Software* (FGS). That is a service architecture with *international application programming interfaces* (IAPIs) that assist software development for global markets. In the FGS, application functionalities have been clustered into six groups of interrelated services. The groups are: basic data structures and algorithms, operating system, windowing system, databases, communication support systems, other applications. An important new concept introduced is LocaleContext, which is associated to internationalised data structures. LocaleContext is also used in various locale specific service calls. As an example, one can consider hyphenation, spelling, and grammar checking of multilingual documents with varying script systems. In the end of the paper there is a case study on applying the principles of the FGS in designing a multi-lingual version of a multi-platform form program, OsiCon Form.

1. INTRODUCTION

In order to reach a large international user base it is necessary that *globalisation* of a software product is done properly. The culturally and linguistically dependent parts of the software must be isolated, a process referred to as *internationalisation*. These parts include e.g. text manipulation and display, character-encoding methods, collation sequences, hyphenation and morphological rules, formats used for numbers and dates, as well as more subtle cultural conventions such as the use of icons, symbols and colour. The local market requirements for these items are encapsulated in the term *locale*.

Localisation is the opposite of internationalisation, taking an internationalised product and adding features to match it to the language and culture of the target market. It does not just consist of translating menus, commands and messages into the required language, but includes adaptations for the culture and business practices of the target country. The term *globalisation* means the whole process starting from feasibility studies, producing an internationalised base version of a software product and then deriving multiple localised versions from it.

The idealised model of internationalisation and localisation is presented in Figure 1. The feasibility study consists of market analysis and analysis of internationalisation options. When existing software is under consideration, it is important to know how international it already is. The feasibility study gives information for decision making in which there are three basic options. The potential new markets may be evaluated to be too expensive considering the

[*] The work was done by VTT Information Technology in the GLOSSASOFT project (LRE 61003) of the Telematics Applications Programme of EC.

[**] Currently on leave at Helsinki University of Technology, Finland.

localisation costs or otherwise unprofitable. If positive decision is made, one may apply direct localisation without internationalisation, or, as we suggest, go through a separate internationalisation phase. For new software, this kind of preparation for international markets should be natural at least at a basic level.

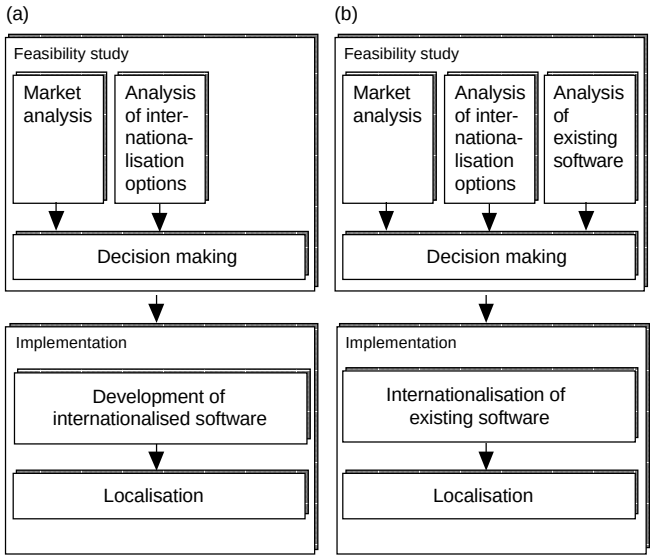

Figure 1. Idealised model of software internationalisation and localisation while (a) developing new software, and (b) dealing with existing software.

Figure 2 describes the relation between the globalisation processes and the classical waterfall model of software engineering. In practice, the phases of the idealised model appear partially in cycles.

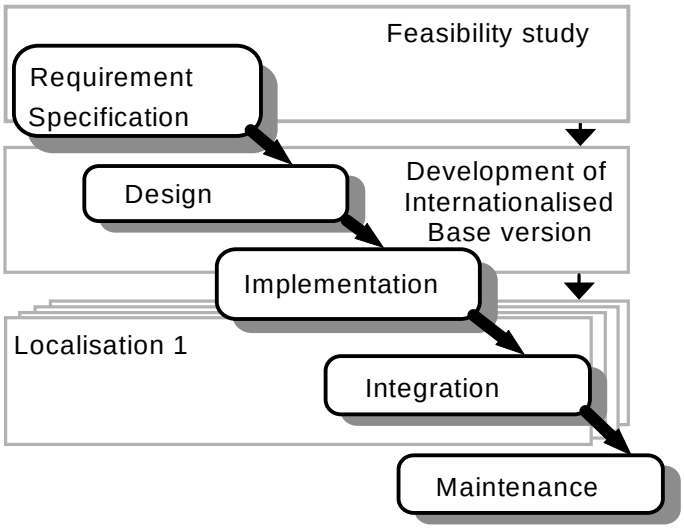

Figure 2. Relating internationalisation and localisation of software to the classical waterfall model of software engineering.

There are several works which cover some or most parts of the globalisation process. Some of those that concentrate on a specific platform are [Apple 92, Carter 92, Digital 91, Microsoft 92, O'Donnell 94, and Ternasky 93], althought many of these cover aspects which are

important in any platform. A more general approach is taken by [Taylor 92, and Uren 93]. Taylor discusses localisation issues rather straightforwardly presenting examples using C. On the other hand, Uren et al. perhaps have their weak points in technical details related to practical programming. [Nielsen, 90] concentrates on user interface aspects. [Yazdani, 93] describes multilinguality in multimedia being in many aspects rather superficial. Software internationalisation and software engineering aspects are the main focus of [Hewlett-Packard 92, and X/Open 93]. A comprehensive collection of Glossasoft project results will appear in [Glossasoft 96].

The purpose of this article is to introduce a service architecture for internationalised software development. The aim is to provide a general framework for practical software engineering which is described in chapter 2. Chapter 3 discusses internationalisation and localisation of a specific software product along the outlined principles. The larger context of this work has been reported in [Hudson 95].

2. THE FRAMEWORK

The Framework for Global Software (FGS) proposes a service architecture with *international application programming interfaces* (IAPI) that assist software development for global markets (Figure 3). In the FGS, application functionalities have been clustered into six groups of interrelated services. Table 1 illustrates these groups. The functionalities covered are listed for each group, and those that involve locale specific behaviour are written in bold face. The IAPI detaches the locale independent kernel of an application from the locale dependent services. In the FGS a new concept, LocaleContext, addresses the problems of parametering locale specific behaviour and realising locale conscious data structures. A LocaleContext is an object which can be defined from scratch or by overriding some properties of an already existing LocaleContext. LocaleContext is used in various locale specific service calls. Hyphenation, spelling, and grammar checking of multilingual documents with varying script systems are examples of some of these calls.

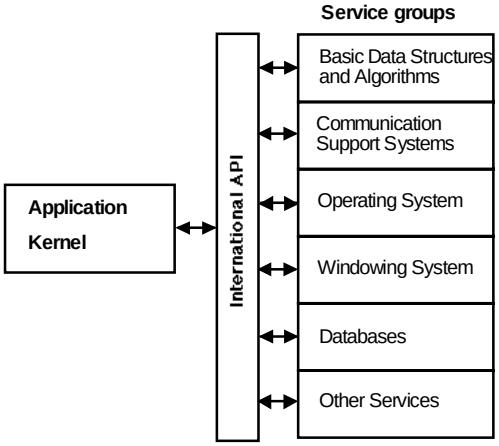

Figure 3. The structure of the FGS.

Basic Data Structures and Algorithms	locale context **characters, strings, streams** **string search, sorting, comparison, case conversion** **spelling, hyphenation** hash tables, lists, containers data compression, data encryption
Communication Support Systems	**messaging interfaces (electronic mail, EDIFACT)** protocol interfaces (sockets, X/Open Transport)
Operating Systems	processes and **interprocess communication** system configuration, memory management, resources (device drivers etc.) **libraries** **file system** **object management support** **basic I/O** **clock and calendar** **command interface**
Windowing System	**windows, views** event handling **controls (including text editing)** **cursors, mouse handling, beep/flash** interclient communication (DDE, drag and drop, clipboard services) **graphics, icons, colours** **fonts, keyboard drivers** **voice and sound**
Databases	**highly structured databases (e.g. relational databases)** **hypermedia databases**
Other Services	**various servers (help engines, dictionaries)**
	document management facilities (SGML, ODA, OpenDoc; indexing facilities)
	security
	debugging support

Table 1. Examples of services in the FGS. Locale dependent ones have been denoted using bold face text.

As an example of internationalised data structures and their usage, Figure 4 illustrates an internationalised string which consists of segments that can be in different locales. Internationalised string processing services are needed in order to process such strings. Windowing system services need to include internationalised controls. An internationalised menu may contain items in different script systems. An internationalised text edit control needs to cope with different script systems in different parts of the document. Writing direction, font, and keyboard map would be selected depending on the LocaleContext.

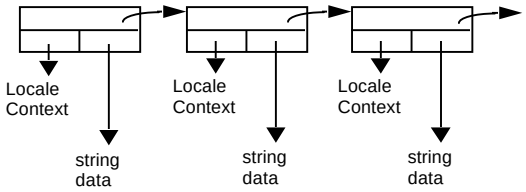

Figure 4. An internationalised string.

The general format of an internationalised control constructor function in the C language could be:

id CreateIControl (Frame &frameGeom, Position &position,
id fatherId, Style &styleStructure, LocaleContext lc);

Related accessory functions would be enhanced to handle internationalised data structures. Their usage could have side effects on the controls due to locale changes. A simple function for setting window titles, **SetITitle(id controlId, IStringId titleId)**, could be defined to effectively change the major locale of the window according to the internationalised input string parameter. This would be reflected in the menus and status bars of the window. These examples demonstrate the semantic richness that may be involved in internationalising different services. Moreover, it gives an idea of how powerful the usage of LocaleContext could be. Depending on the ambitiousness of its implementation a LocaleContext could control:

- **Language related issues:** alphabet, primary writing direction, month and day names and abbreviations, ordinal abbreviations, spell-checking, hyphenation, and other linguistic aids etc.
- **Text processing functionality:** character sets and associated fonts, conversion functions (upper/lowercase, diacritical/accent removal between character sets), collating sequences etc.
- **User interface text and control:** error, help, system texts, flow control, command parsing tables, recognition logic for commands (localised command languages), searches etc.
- **Country:** keyboard control (key sequence-to-function mapping, keyboard drivers), other device control (e.g. print control mapping), time transformation (calendar offset from GMT, zone name, zone abbreviation, daylight savings time), currency formats (currency symbol, negative currency indicator), data representations (decimal separator, thousands separator, list separator, default formats for time, date, phone number, and addresses) etc.

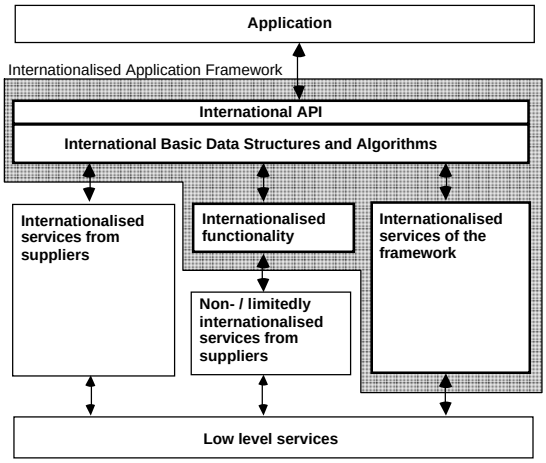

Figure 5. Implementation choices for IAPI services.

Figure 5 illustrates the three ways in which the IAPI services can be implemented on widely used platforms. First, for services which are not provided by suppliers, the services themselves must be (re)implemented providing internationalisation support. Implementation is based on low level, non-internationalised, system services. For example, the internationalised string input function needs to use lower level services of the operating environment. This function needs controlling the keyboard maps according to the locale context information and transforming the keyboard input in the form of an internationalised string.

Secondly, there are services provided by suppliers, but which are not (adequately) internationalised, internationalisation related functionality must be handled in the framework code. Existing services are then used to provide the actual functionality. For example, internationalised button control in the GUI part of the IAPI can be implemented by properly parametering the existing button components of the operating environment. In its simplest form, it is only needed to select the right font and orientation for the capture text of the standard button component.

The third and simplest case is when already existing internationalised services can be applied straight via an interface that wraps the syntax of the calls to conform the IAPI. For instance, Microsoft provides with its office programs an internationalised spelling checker that has its own well defined API called CSAPI. The IAPI can access these spelling services simply via this API.

The FGS can be used as a model of how an internationalised application could be made to interface to external services, how locale-specific features could be parameterised, and how locale consciousness could be handled in data structures. The IAPI of the FGS provides guidelines and examples on how and where the borderlines could be set, when language and culture-dependent parts are detached from those parts of an application that are universally applicable. Standards, such as Unicode [Unicode 90, 91, and 92] and C language internationalisation support extensions by X/Open [X/Open 1993] provide good basis for developing an IAPI. An extensive presentation of the FGS will be included in [Glossasoft 96].

3. INTERNATIONALISATION AND LOCALISATION OF OSICON FORM

The internationalisation and localisation process of OsiCon Form[1] (shortly OcForm) program was used as a Glossasoft case study to see how well the principles of the FGS suit to everyday software engineering. OcForm provides a graphical user interface for presenting and editing contents of EDIFACT messages, which are widely used in various interorganisational business or administrative transactions. OcForm is a multi-platform application. The technical realisation is based on C++/Views 2.0 library product of Intersolv Inc. This library provides an abstracted high-level API that has been implemented on many platforms (UNIX/X Window System, MS Windows (16 & 32 bits API), Macintosh, OS/2 etc.).
OcForm required internationalisation in three respects:

- GUI controls that are fixed in the program should be dynamically adjusted to fit the needed locale,
- layouts and text definitions of a form should be storable for many locales and the right definitions should be retrieved according to the choice of locale made by the user, and
- form data should be partly internationalised by providing table based translation facilities for some of the form elements. These elements have locale independent internal data values that have locale dependent external presentations.

Internationalisation required also the design of facilities for dynamic change of the locale of the form. An active application must be able to have simultaneously different locales in different windows with respect to the locales of the forms. The internationalisation solutions should be transportable across multiple platforms. Figure 6 illustrates the outlook of the result that satisfied the requirements.

The internationalisation was constrained by the following limitations in the used high level multi-platform API:

- no support for multiple keyboard maps (keywise mapping can be adjusted),
- no support for 16 bit character sets (type wchar, Unicode tables),
- no support for locale specific string manipulation (e.g. collation),
- no support for locale specific representations (e.g. date), and
- no platform independent fonts.

Due to the limitations in the tools the character set support was decided to be restricted to ISO 8859-x character sets, thus targeting to Western European and American markets.

The internationalisation process consisted of five major tasks:

- separating the constant text templates from the program structure and designing a table based storage for storing locale specific versions of the texts,
- adding new parameters to the program to hold locale specific information such as fonts, icons etc.,
- adding to certain internal data structures an association to a LocaleContext object and defining rules for updating, storing, loading and dynamically changing locales,
- enhancing the form definition language to include an association to LocaleContext and enhancing the form definition retrieval system to use LocaleContext in selection, and

[1] Software product of VTT Information Technology.

- designing algorithms for dynamic changing of locale of the user interface.

Next, there will be a short description of each major task.

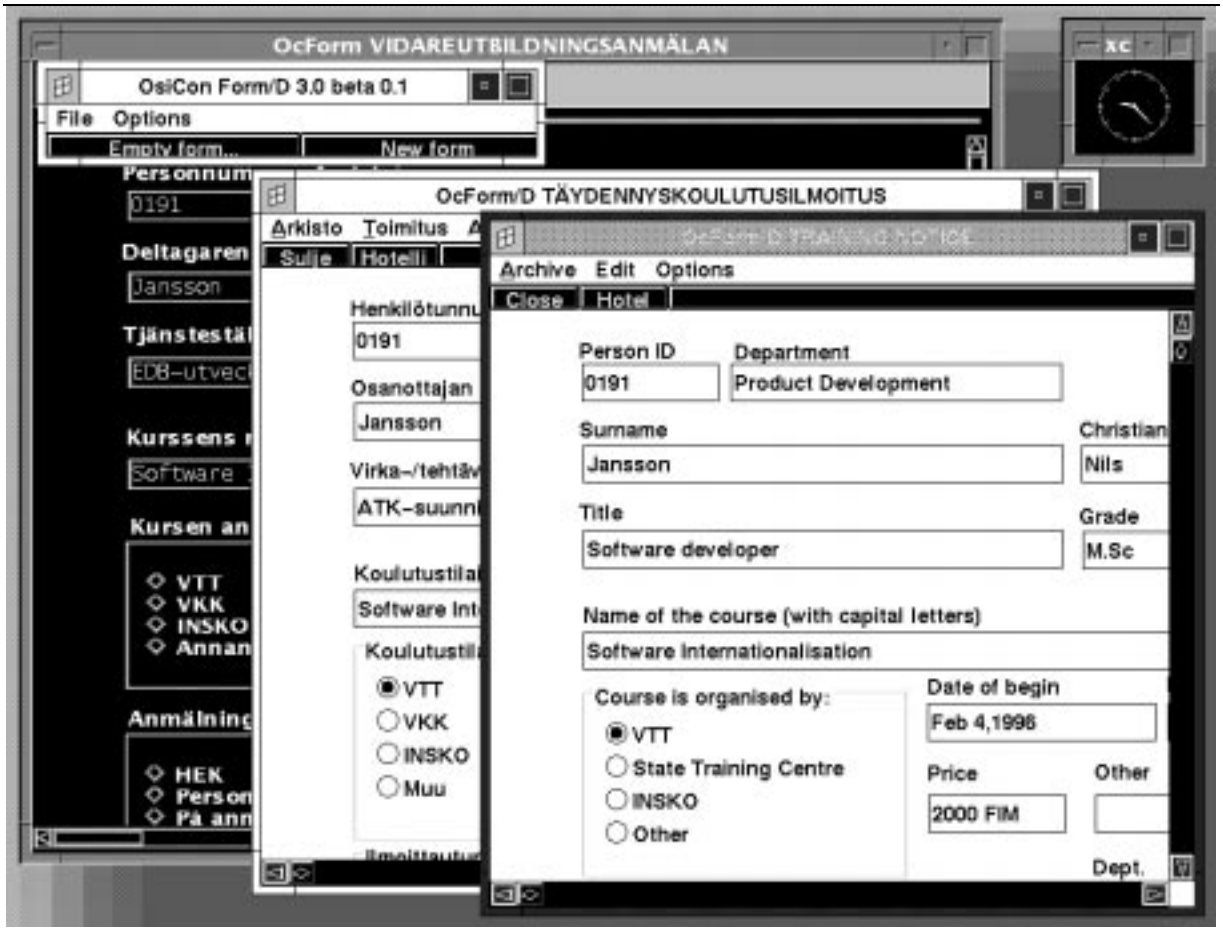

Figure 6. The outlook of OcForm after internationalising it and localising it to English, Swedish, and Finnish locale. In the back, the HP-UX/Motif version is executed with a training notice form in Swedish. In the front, the MS Windows version is executed on the same UNIX machine under WABI Windows emulator, and with both Finnish and English locales.

3.1 Internationalising texts

As far as cross-platform multi-lingual text resource mechanisms were not available, we decided to base the storage and retrieval of static user interface texts on text tables and internationalised output functions. The constant strings and string templates (e.g. C-language stdio-library-style of format-strings) were separated from the program code into a simple message catalogue file, where the translations were added as well. For automating this phase of work we made a small string extraction and substitution program. The next insert shows how the message file looks like after adding English and Swedish translations into it:

```
defTxt(WPARAMS_CPP_1,
      "%s -tiedostossa ei ole [OcForm]-lohkoa, jossa olisi parametri %s.",
      "%s file does not contain a [OcForm] block, with the parameter %s.",
      "I filen %s saknas ett [OcForm]-block med parametern %s.");
defTxt(ADLGMTRX_CPP_1,
      "Valitse orientaatio:",
      "Choose orientation:",
      "Välj riktning");
```

When the strings were separated from the code, they were substituted with an expression containing a symbol that may be defined to be either a C language pointer variable or a symbolic constant of C pre-processor. For instance the string "Valitse orientaatio" was substituted in the code with the expression `getLocalString`(ADLGMTRX_CPP_1). The final association of one version of the string to its reference in the source code is defined by `defTxt` and `getLocalString` macros. The choice of the association mechanism is made where `defTxt` and `getLocalString` are defined. The association may be static and carried out at compile time or fully dynamic at run time. The choice depends on the

We are interested in the ability to dynamically select the correct string for the locale. We need to have all the strings accessible in the running program. Thus `getLocalString` macro is defined to build a function call to a function that gets as its arguments a pointer to a table of strings and a LocaleContext object, and returns a string pointer. This function is redefined by each object class when necessary, in order to implement appropriate scoping rules to locale settings.

We also made experiments related to the on-line generation of messages using a morphological generator and message templates, which have been extended with parameterised calls to word-form generation services. As the generator we used a Finnish morphological generator named FINGEN (FINnish GENerator), which is a product of Lingsoft Ltd, Finland [Lingsoft 94]. The generator is not included in the baseline system. The results of the experiments and some alternative ways for implementing dynamic message generation are reported in [Spyropoulos 1995].

3.2 Locale Specific Parameters

The programs utilise MS Windows type of initialisation files for reading their parameters. We decided to include to the parameters locale specific user interface fonts, program icons and form definition paths. Increasing the set of locale specific parameters in the future is simple. Each locale has its own parameter block starting with the header [*locale_name*]. In order to preserve the portability of the program, we implemented our own facilities for accessing and updating initialisation files, and thus made them transportable.

3.3 Locale Information Handling in Object Structures

One of the goals in the internationalisation process was to change the design of the original software so that the locale of its user interface can be dynamically changed and that there may be simultaneously open forms that have different locale. In order to reach this goal we had to decide how to distribute locale information along the data structures and GUI elements. We decided to associate a LocaleContext object to appropriate objects. This object would then be passed as an extra argument to all functions that implement localised features. E.g. functions that retrieve messages from message catalogues receive it.

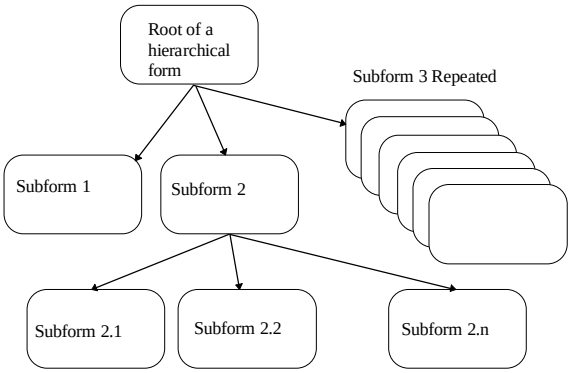

Figure 7. An example of hierarchical forms.

The form we are dealing with is shown to the user as a hierarchical tree of GUI windows between which the focus may be freely switched in the user dialogue (Figure 7). Any window may be given property "repeated" which in practice means, that the window makes an interface to a set of equally structured data records that may be browsed forward and backward with "<<" and ">>" buttons.

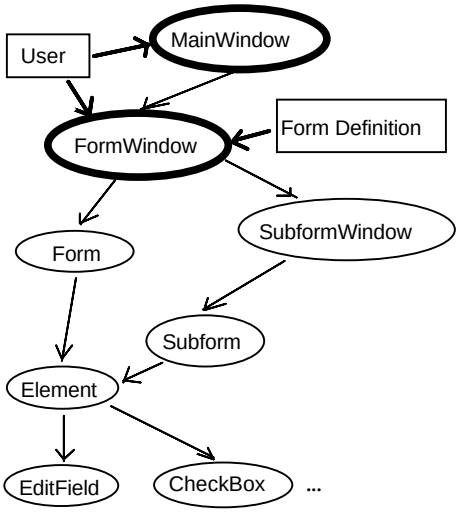

Figure 8. Passing through LocaleContext information in form objects.

Figure 8 describes how locale information is handled. The ellipsoids present objects classes within the programs. Objects of the bolded classes store locale specific information. The main window (MainWindow) and root windows of forms (FormWindow) both store LocaleContext objects. Locales of other objects depend on these stored locales and are solved along the arrows.

When the program is started the MainWindow is initialised in the default locale given as one of the initialisation parameters. The user may change the locale by choosing *Lingua* from the *Options* menu. If the form definition that the user decides to open contains a locale code associating it with a specific LocaleContext then the corresponding set of windows will be opened with menus and other user interface elements localised into that locale. If locale code is missing from the form definition then the form is opened in the locale the MainWindow is currently having. Subform windows always inherit their locale from the root window of the form.

3.4 Locale Information in Form Definitions

The form definitions were extended to carry locale codes. Moreover, we organised the relationships of form definitions of a single form so, that one of the definitions is the default definition. This definition contains links to the definitions of other locales. When a form is opened the program first looks into the default definition and then into all those definitions that are linked to the default definition. If a definition is not found with the same locale as the main window, the form will be opened with the locale of the default definition.

3.5 Dynamic Changing of Locale

Changing the locale of the main window is carried out by reinitialising all its GUI elements with new fonts and icon specified in the initialisation parameters and new strings retrieved from the message table. C++/Views library provides good facilities for changing the elements and the change happens elegantly in place. The change of the main window locale is not percolated to those form windows that are already open. Change of the locale of a root window of a form causes all its child windows to be closed and the root window itself to be first cleared and the rebuilt according to the form definition that has been designed for the new locale. This dynamic locale change is not visually as elegant as with the main window. As the layouts, formats of data etc. may differ a lot depending on the locales, this seemed to be the best choice. The form data is fully preserved in the locale change.

4. CONCLUSIONS

The case study with the OsiCon Form proved the soundness of the four central design principles of the Framework for Global Software:

1. An internationalised application consists of an application kernel that has been designed in locale independent manner.
2. The application kernel can use internationalised external services via an IAPI.
3. An IAPI (International Application Programming Interface) lets parameter locale specific behaviours of external services using LocaleContext structures/objects.
4. Locale dependent data structures are made locale conscious by associating (statically or dynamically) LocaleContext structures to them.

Currently there are no comprehensive industry standards available for an IAPI. Fortunately, standards with limited coverage are available (e.g. Unicode, NLS of X/Open) and can be used to implement parts of an IAPI. In everyday work, solutions have to be based on the available APIs provided by systems software and licensed third party tools such as database access and GUI libraries. Many internationalised services can be implemented by introducing extra mapping layers on top of the existing services. However, some of the services are beyond this layering approach and need profound functional extensions to become internationalised (e.g. an internationalised text edit control).

ACKNOWLEDGEMENTS

The author wishes to thank Prof. Pat Hall, Ray Hudson, Dr. Costas Spyropoulos, Dr. Evangelos Karkaletsis, George Vouros, Dr. Stavros Kokkotos, Francis Magann, Nikos Stamatakis, Rafik Belhadj, Prof. Seppo Linnainmaa, Sakari Kalliomäki, Krista Lagus, Tuula Käpylä and all who have been helping to realise the reported work.

REFERENCES

Apple Corporation, 1992. *Guide to Macintosh Software Localisation,* Addison-Wesley Publ. Company, May 1992.

Carter, D.R., 1992. *Writing Localizable Software for the Macintosh*. Addison-Wesley, 1992.

Digital Equipment Corporation, 1991. *Digital Guide to Developing International Software*. Digital Press, 381 p.

Farid, M. 1990. *Software Engineering Globalization and Localization*. 1990 IEEE International Conference on Systems, Man and Cybernetics, Conference Proceedings, pp. 491-495.

Glossasoft Consortium, 1996. *Software Design for Globalisation: A Practitioner's Guide,* John Wiley and Sons (to be published in Spring 1996).

Hewlett Packard, 1992. *How to Develop Internationalised Software*. Hewlett Packard.

Hudson, R., Lehtola, A., 1995. *GLOSSASOFT: Methods and Guidelines for Software Internationalisation and Localisation.* Proceedings of the Language Engineering Convention, London, October 1995, 8 p.

Lingsoft Ltd., 1994. *FINGEN Reference Manual*.

Microsoft Corporation, 1992. *The Windows Interface: International Handbook for Software Design*. Microsoft Press, 239 p.

Nielsen J (ed.) 1990. *Designing user interfaces for international use*. Elsevier, 230 p.

O'Donnell, S.M., 1994. *Programming for the World*. Prentice Hall.

Spyropoulos, C.D., Karkaletsis, E.A., Vouros, G.A., Honkela, T., Lagus, K., and Lehtola, A., 1995. *Investigating On-Line Message Generation in Software Applications: The GLOSSASOFT Results.* ERCIM Workshop on "Towards User Interfaces for All: Current Efforts and Future Trends", Heraklion, Greece, October 1995, 13 p.

Taylor, D., 1992. *Global Software*. Springer-Verlag, 319 p.

Ternasky, J. & Ressler, B.K., 1993. *Writing localizable applications.* Apple Technical Journal, Issue 14, pp. 7-33.

Unicode Consortium, 1990. *The Unicode Standard, Version 1.0*. Volume 1, Addison-Wesley, 682 p.

Unicode Consortium, 1992. *The Unicode Standard, Version 1.0*. Volume 2, Addison-Wesley, 439 p.

Unicode Consortium, Caldwell, J.T., 1991. *Unicode*. in Characters and Computers, Main V.H. and Liu Y. (ed.), IOS Press.

Uren, E., Howard, R., Perinotti, T., 1993. *Software Internationalisation and Localisation - An Introduction*. Van Nostrand Reinhold, 300 p.

X/Open Company, 1993. *Internationalisation Guide Version 2*. X/Open Company Ltd., 180 p.

Yazdani, M., 1993. *Multilingual Multimedia*. Intellect, 210 p.