(12) INTERNATIONAL APPLICATION PUBLISHED UNDER THE PATENT COOPERATION TREATY (PCT)

(19) World Intellectual Property Organization
International Bureau

(43) International Publication Date
7 August 2003 (07.08.2003)

PCT

(10) International Publication Number
WO 03/065177 A2

(51) International Patent Classification⁷: G06F

(21) International Application Number: PCT/US03/03151

(22) International Filing Date: 3 February 2003 (03.02.2003)

(25) Filing Language: English

(26) Publication Language: English

(30) Priority Data:
60/353,487 1 February 2002 (01.02.2002) US

(71) Applicant and
(72) Inventor: FAIRWEATHER, John [US/US]; 1649 Wellesley Drive, Santa Monica, CA 90405 (US).

(74) Agents: THIESSEN, Kendall, I. et al.; Gibson, Dunn & Crutcher LLP ,, 1801 California Street, Suite 4100, Denver, CO 80202 (US).

(81) Designated States *(national)*: AE, AG, AL, AM, AT, AU, AZ, BA, BB, BG, BR, BY, BZ, CA, CH, CN, CO, CR, CU, CZ, DE, DK, DM, DZ, EC, EE, ES, FI, GB, GD, GE, GH, GM, HR, HU, ID, IL, IN, IS, JP, KE, KG, KP, KR, KZ, LC, LK, LR, LS, LT, LU, LV, MA, MD, MG, MK, MN, MW, MX, MZ, NO, NZ, OM, PH, PL, PT, RO, RU, SD, SE, SG, SK, SL, TJ, TM, TN, TR, TT, TZ, UA, UG, US, UZ, VN, YU, ZA, ZM, ZW.

(84) Designated States *(regional)*: ARIPO patent (GH, GM, KE, LS, MW, MZ, SD, SL, SZ, TZ, UG, ZM, ZW), Eurasian patent (AM, AZ, BY, KG, KZ, MD, RU, TJ, TM), European patent (AT, BE, BG, CH, CY, CZ, DE, DK, EE, ES, FI, FR, GB, GR, HU, IE, IT, LU, MC, NL, PT, SE, SI, SK, TR), OAPI patent (BF, BJ, CF, CG, CI, CM, GA, GN, GQ, GW, ML, MR, NE, SN, TD, TG).

Declaration under Rule 4.17:
— *of inventorship (Rule 4.17(iv)) for US only*

Published:
— *without international search report and to be republished upon receipt of that report*

For two-letter codes and other abbreviations, refer to the "Guidance Notes on Codes and Abbreviations" appearing at the beginning of each regular issue of the PCT Gazette.

WO 03/065177 A2

(54) Title: SYSTEM AND METHOD FOR NAVIGATING DATA

(57) **Abstract:** The present invention provides a dynamic hyperlinking architecture that permits a user to enable/disable hyperlink domains that are automatically applied to every bit of textual data present in the system or displayed to the user. The present invention includes synchronous and asynchronous, inter-thread function calls, including support for function overrides in a threaded scope dependent manner. The present invention also supports broadcast (multiple call) call configurations and run-time examination of function registries. In the preferred embodiment, the system comprises a threaded environment, threaded type dependant symbolic functions and a hyperlinking system uses both the threaded environment and symbolic functions to dynamically create links to data and functions that are displayed and/or executed responsive to user selection of a link.

SYSTEM AND METHOD FOR NAVIGATING DATA

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

A user interface is only as good as the focus that it provides. Digital information environments, such as the World Wide Web, are designed to capture and lead the focus of the person using them. This is often based on the agenda of the person creating the web page and most frequently that agenda is to garner advertising dollars. Thus, the problem of searching for the answer to something on the web only to be forced to focus on irrelevant web sites is a common experience. In such a scenario, a user often fails to find what they were looking for, often forgetting what they were looking for in the first place. This effect occurs because the digital domain is not constrained by the same relevance falloff law that constrains the analog world. Each navigation step may be arbitrarily large, and the human mind is poorly equipped to maintain focus, and thus the search for meaning or relevance in this environment is very difficult. Nowhere is this problem more inherent than in the use of hyperlinks.

In any large collection of disparate data, effective navigation becomes critical. For example, on the Internet the approach taken to navigation was to implement embedded "hyperlinks" which transition the user's focus to the URL referenced in the hyperlink. This works effectively, but is a manual, restrictive, and error prone business. The web-site designer must manually insert the chosen hyperlink to the URL, thereby enforcing his perspective on the user, rather than the perspective of the user. Worse yet, URLs change continuously and the referencing link then becomes out of date and useless. What is needed, then, is the ability to define and enable/disable hyperlink domains on a per-user basis based on the information and world-view that he, or the organization of which he is a member, brings to the problem the user is researching. In other words, in addition to conventional hyperlinks, which reveal the focus of others, what is needed is a user-centric, organization-centric, and domain-centric hyperlinks that are automatically applied to every bit of textual data present in the system or displayed to the user.

SUMMARY OF INVENTION

The present invention provides such a system. The present invention provides a dynamic hyper-linking architecture under the control of each user, not under the control of the information source. The present invention includes synchronous and asynchronous, inter-thread function calls, including support for function overrides in a threaded scope dependent manner. The present invention also supports broadcast (multiple call) call configurations and run-time examination of function registries. In the preferred embodiment, the system comprises the following:

- A threaded environment providing the following abilities:
  - a) Association of arbitrary data, in this case function registries, with threads;
  - b) Hierarchical nesting of thread contexts with corresponding UI context relationships;
  - c) Ability to pass 'events' containing messages between threads;
  - d) Environment supplied transparent invocation of certain events;
  - e) Ability to 'look-up' threads based on a unique thread/widget ID;
- A series of function registries associated with each context in the system, including a global registry whose scope encloses that of all others. Within these registries, using API calls, functions can be registered by name (as a text string) by specifying the ancestral scope at which the registration should occur; and
- In the preferred embodiment, an API that permits execution of functions by name that internally searches the relevant thread's registries in an order determined by gradually widening scope (as defined by the threaded environment) and which causes the necessary functions to be executed, with the parameters supplied by the caller, either in the calling context ('near') by direct call, or in the registering context ('far') by call in response to an appropriate event. A 'reply' function may also be specified which allows function results to be returned to the calling context in a synchronous or asynchronous manner.

Furthermore, the present invention provides a system for implementing threaded type-dependant asynchronous invocation of a set of named logical actions in thread dependant, scoped, manner including support for overriding the invoked functionality within any scope, passing of arbitrary parameters from invoker to invoked, type ancestry dependant inheritance of invocation behaviors (including scope dependency) based on a threaded symbolic registry scheme such as described above. Finally, a hyperlinking system uses these features to dynamically modify a user interface such that any text in a user interface can be hyperlinked to one or more sets of types data using hyperlink dictionaries that may be user defined or global. Additionally, clicking on such a hyperlink can invoke one or more functions (as described above) based on the scope of the functions permitting display of wide ranging data and media types.

It is anticipated that further modifications and extensions will also be provided. For example, the system could be extended to support the ability through API calls to associate arbitrary data and logical flags with registered functions. Additionally, they system could be extended to support the ability to inhibit/enable functions in the registry(s) by scope through described API calls.

BRIEF DESCRIPTION OF THE FIGURES

[NONE]

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The technology described herein preferably takes advantage of a number of other key technologies and concepts. Ideally, the reader would be familiar with the technology described in the patent applications listed below in order to fully understand breadth and uniqueness of the present invention. For these reasons, the following technologies, which have been previously described in the following related patent applications, have been fully incorporated herein:

1) Appendix 1 – "Lexical Patent"
2) Appendix 2 - "Memory patent"
3) Appendix 3 – "Types Patent"
4) Appendix 4 – "Ontology patent"

It is important to understand that the invention described herein can be added to any information accessed by the user regardless of source, internal or external. While its application will be described with reference to web pages for simplicity, this is but one example of its application and should not be construed as a limit to the scope of the present invention. The present invention directly addresses the loss-of-focus issue described above by allowing the user to define and modify his or her own hyper-linking environment and allows all of the knowledge of the user or the user's organization to be used to analyze and modify the appearance of the information being displayed. The architecture, within which the user performs his daily activities, and the user interface (UI) it presents, provides and automates this facility. More specifically, when a hyperlink is clicked, the architecture identifies the nature, type and location of the datum to which that hyperlink refers. Once the datum type has been retrieved, the architecture automatically launches the appropriate display behaviors to show the target datum to the user in the most appropriate manner, which in many cases will be context dependant.

The present invention is built up in three layers. The first layer (as exemplified by the API calls starting with OC_) is targeted at the more general problem of symbolically invoking functionality within a complex threaded environment in a manner that permits both local and remote synchronous and asynchronous function invocation and customization of the actual functionality invoked in a context sensitive and scope dependant manner. The second layer (as exemplified by the API calls starting with DB_) ties this capability to a type-dependent, ontology-based invocation system. The third layer provides the capabilities required to handle and display ontology-centric hyperlinks.

Threaded Symbolic Function Calls

The first layer provides functionality that permits threaded, scope dependant symbolic function invocation. Specifically, the first layer allows function calls to be made between and across threads in a symbolic, possibly asynchronous manner. Throughout this discussion, threads will be referred to as 'widgets' where each widget in the system has a unique widget ID that can be used to reference it.

As an initial matter, it is helpful to describe the preferred thread architecture of the substrate within which the functionality described herein is intended to run, and which confers the ability to represent nested scope. Other substrate architectures are possible provided that they support at least some portion of the scope behaviors described herein. The need for scope dependant configuration of invoked functionality, and its complete divorcing from the consideration of the invoker, permits large complex systems to be easily assembled out of flexible adaptable building blocks. This is a problem that is poorly handled by more conventional approaches such as object-oriented programming, for example. While such approaches could be used, this is not the preferred approach.

The following description refers to compiled, executable code as 'atomic widgets'. Atomic widgets may be combined and nested within higher-level widgets (that generally do not contain executable code) and are referred to as 'compound widgets'. Collectively, atomic widgets and compound widgets will be referred to as 'widgets'. In addition to logical nesting within compounds, the present invention also provides a corresponding layout of widgets within the user interface (UI) implied by such nesting. Compound and atomic widgets may be combined into higher-level compound widgets to an arbitrary number of levels. In the preferred embodiment, widgets can be grouped into loadable and executable 'applications', comprised of one or more (possibly nested) widgets, which are known as 'views'. Generally, there will be one or more windows within the user interface that correspond to a given view. Views in turn can be combined into logical groups of views known as view packs. Further, any widget within a view or view pack may cause the launching of another view or view pack, and the launch dependency between these various views in the system is tracked and utilized as part of determining 'ancestry'. Thus, we have the concept of a scope or ancestry chain for any given widget context running in the system that contains some or all of the elements depicted below:

```
Global Environment context
    View Pack
        View
            Launched View [Pack]              -- may be nested to any # of levels
                View
                    Compound Widget           -- may be nested to any # of levels
                        Atomic Widget
```

Because there is a close match between UI window layout and the logical nesting of widgets described above, this ancestry chain closely matches the perceived visual context of any given widget. This approach permits use of the scope defined by the ancestor chain to configure the behaviors and resultant appearance of invoked functions into the context from which they are invoked. For simplicity, the current widget's scope will be defined to be zero on a signed number line. Increasing widget ancestry can then be referenced as +1 for the parent, +2 the grandparent etc. This positive incrementing continues until the nesting within a given view is exhausted. The ancestry is also defined in the opposite direction. For example, switches to –1 (local view scope) and increases in the negative direction with –2 being view pack scope, -3 launching view scope (if any), and so on in the negative direction until the chain runs out. Finally, global environment scope within which all other scopes are defined can be reference using the constant –32768.

In the preferred embodiment, the implementation of symbolic function registries in the present invention utilizes string lists (as described in the Memory Patent) to store the information passed on the call to OC_RegisterFunction(). Each scope node discussed above may have such a registry associated with it if any functions have been registered. A such, the present invention access these registries and looks for registered function in expanding scope order during a call to OC_CallSymbolicFunc(). The basic scope logic is implemented by the internal function OC_SymbolicFuncLoc() the pseudo-code for which is given below:

```
static ET_StringList OC_SymbolicFuncLoc(              // obtain function
address list
```

```
                int32               aWidgetID,        // I:Widget ID(0 =
current)
                int32               *aScopeID,        // O:scope widget ID
                int32Hdl            *index,           // O:~0 term. match
index list
                charPtr             aFuncName,        // I:symbolic
function name
                int32               options,          // I:various logical
options
                int32               aMatchWidgetID,   // I:matching widget
ID,or 0
                ET_SymbolicFunc     aMatchFuncAddr    // I:Matching fn.
address or NULL
                              )                       // R:String List or
NULL
{
   if ( aWidgetID == kGlobalSCOPE )
      scopeWP = 0;
   else
   {
      scopeWP = convert aWidgetID & aMatchWidgetID to reference
      vh = view handle of scopeWP
   }
   myIndex = -1;
   if ( aWidgetID != kGlobalSCOPE )
      while ( !ctr && scopeWP )                       // search widget's
ancestry chain
      {
         if ( aScopeID ) *aScopeID = scopeWP->widgetID;
         sL = scopeWP function registry;
         if ( sL )
         {
            do
            {
               myIndex = search sL for name specified
               if ( name found )
               {
                  if ( !(options & kIncludeSuppressed) )
                     if ( function suppressed )       // check ! supressed
                        continue;
                  extract all required values
                  add myIndex to *index array
               }
            } while (myIndex >= 0);
         }
         if ( !ctr )
         {
            scopeWP = parent widget of scopeWP
            if ( !scopeWP )                           // ran out of
widgets!
            {
               if ( in a view pack )                  // now work through
views...
                  scopeWP = view widget of prime view of pack
               else if ( this view was launched by another )
               {
                  scopeWP = view widget of the launcher
               } else scopeWP = 0;
            }
         }
      }
   if ( !ctr && !(options & kNoGlobalSearch) )        // try the global
registry...
   {
      if ( aScopeID ) *aScopeID = 0;
      sL = global registry
      myIndex = -1;
      if ( sL )
      {
         do
         {
            myIndex = search sL for name specified
            if ( name found )
            {
               if ( !(options & kIncludeSuppressed) )
                  if ( function suppressed )          // check ! supressed
                     continue;
               extract all required values
               add myIndex to *index array
            }
         } while (myIndex >= 0);
      }
   }
   if (, !ctr )
      sL = NULL;
   return sL;
}
```

In this embodiment, the function above returns a string list containing all matching functions registered at the relevant scope. From this information, the implementation of most routines in the function registry API can be deduced. For example, one implementation of the function OC_CallSymbolicFunction() is as follows:

```
Boolean OC_CallSymbolicFunction(                       // call a symbolic
function
               charPtr          aFuncName,             // I:symbolic
function name
               void             *aFuncParameter,       // I:parameter (or
NULL if N/A)
               ET_SymbolicReply aReplyFunc,            // I:Address of reply
fn. or NULL
               int32            aMatchWidgetID,        // I:Matching widget
ID or 0
               ET_SymbolicFunc  aMatchFuncAddr,        // I:Matching fn.
address or NULL
               int32            options                // I:Various logical
options
                              )                        // R:TRUE for success
{
   sL = OC_SymbolicFuncLoc(0,NULL,&index,aFuncName,...);
   if ( !sL || !index ) return NO;
   i = count the matches returned
   if ( !i ) return NO;                                // no functions found
   ofP = NULL;
   for ( i-- ; i >= 0 ; i-- )                          // call fn. for every
element
```

```
   {
      wid = 0;
      sP = get function address
      if ( sF )
      {
         wid = get widget ID
         farFunc = near or far call?;
         id = current widget ID
         if ( wid == id )                         // both widget IDs
the same
            farFunc = NO;
         if ( farFunc )                           // call far in
original context
         {
            ffP = (OC_FarFuncDescPtr)allocate heap pointer
            ffP->func = sF;
            if ( ofP ) ofP->nextFunc = ffP;       // build up a doubly
linked list           ffP->prevFunc = ofP;
            ffP->options = options;
            strcpy(ffP->name,aFuncName);
            ofP = ffP;
            ffP->reply = aReplyFunc;
            ffP->aFuncParameter = aFuncParameter;
            post wake message to registerer's context referencing ffP
            aFarFunc = YES;
         } else                                   // near functions
called here
         {
            (sF)(aFuncName,aFuncParameter,id,options);// call it 'near'
            if ( aReplyFunc )                     // call the reply fn.
               (aReplyFunc)(aFuncName,aFuncParameter,id,options);
         }
      }
   }
   if ( !aFarFunc && aFuncParameter && !(options & kNoParameterDelete) )
      dispose of (aFuncParameter);                // if no far funcs,
delete
   return YES;
}
```

In the wake event handler for a far function, the logic may be implemented is as follows:

```
static void OC_FarFunkWake    (                          // far function
wake handler
               ET_NfyRecordPtr        theWakeEvent       // I:The wake
event record
                              )                          // R:void
{
   ffP = (OC_FarFuncDescPtr)extract from theWakeEvent
   if ( !ffP ) return;
   lastGuy = !ffP->nextFunc && !ffP->prevFunc;           // are we the last
function?
   if ( ffP->func )
      (ffP->func)(ffP->name,ffP->aFuncParameter,...);    // call symbolic
function
   if ( lastGuy && !ffP->reply && ffP->aFuncParameter &&
        !(ffP->options & kNoParameterDelete) )
      dispose of(ffP->aFuncParameter);                     // de-allocate if
no reply
   if ( ffP->reply )
   {
      ffP->func = ffP->reply;
      ffP->reply = NULL;
      post wake message back to caller's context referencing ffP
   } else
   {                                                       // remove from
linked list
      if ( ffP->nextFunc ) ffP->nextFunc->prevFunc = ffP->prevFunc;
      if ( ffP->prevFunc ) ffP->prevFunc->nextFunc = ffP->nextFunc;
      dispose of(ffP);
   }
}
```

The code above is simply one embodiment of a process for achieving this result. Namely, retrieving functions registered at a given scope and calling the symbolic function as appropriate. As explained above, this functional layer provides threaded asynchronous function calling behavior.

Threaded Type Dependant Invocation

In the preferred embodiment, the symbolic function capability described is extended to a type and ID dependent form suitable for use in an abstract type-dependent invocation scheme. This approach would preferably use a run time accessible type system (a methodology for 'typing" data) and corresponding system ontology. In the preferred embodiment, the run time accessible types system is the types system described in the Types Patent and the system ontology is the ontological framework described in the Ontology patent. Other embodiments, however, could also be used to used.

With a types system and ontology in place, the type-less symbolic functions can be extended to a strongly typed action dependant form by taking advantage of the fact that function names are strings. Specifically, by adding a type dependent wrapper layer (the DB_ calls described below), type names and unique ID numbers can be converted into unique symbolic function names by using the C programming language sprintf() function. For example, the internal symbolic name for an invoker for the action “myAction”, on the type “MyType” having unique ID number “1234” would be “myActionMyType1234”. This form corresponds to what is internally registered by the function DB_OverrideForTypeAndItem(). The corresponding form for DB_OverrideForType() would be “myActionMyType”. Implementation of the other DB_Override…() style functions in the API follows directly from this approach. Using the definition of the invocation record type ET_DBInvokeRec (given below), the basic logic for the function invocation function (DB_Invoke()) could be implemented as follows:

```
ET_ViewHdl DB_Invoke         (                          // Invoke by type and
action
                OSType              aDataType,          // I:Key Data type
                charPtr             actionName,         // I:Action name or
NULL
                ET_DBInvokeRecPtr   iR,                 // IO:The invoker
record
                int32               options             // I:Various logical
options
                             )                          // R:non-zero for
success, or NULL
{
   dT = aDataType;
   if ( !iR->dataType )
      iR->dataType = aDataType;
   if ( aDataType )
   {
      dp = resolve data type(aDataType);                // check we know the
data type
      while ( !dp )                                     // nothing specific
try ancestors
      {
         tid = TM_KeyTypeToTypeID(dT,NULL);             // get ancestral key
type
         if ( tid )
            tid = TM_GetParentTypeID(NULL,tid);
         if ( !tid )
            return NULL;
         dT = TM_GetTypeKeyType(NULL,tid);
         dp = resolve data type(dT);
      }
      iR->options |= kIsClientServerInvokation;
      aDataType = dT;
   }
   if ( !actionName )
   {
      if ( !iR->action[0] )
         strcpy(iR->action,"Display");
      actionName = iR->action;
   } else
      strcpy(iR->action,actionName);

   stillLoop = YES;
   while ( stillLoop )
   {
      stillLoop = NO;
      strcpy(fullName,actionName);                      // first look for
desired form
      if ( dp && !iR->dataItemType[0] )
         strcpy(iR->dataItemType,dp->name);
      strcat(fullName,(dp) ? dp->name : iR->dataItemType);
      strcpy(nameWithID,fullName);                      // form is
'DisplayMyDataTypeName'
```

```
      sprintf(tmp,"%lld",iR->anItemID.id);
      strcat(nameWithID,tmp);                          // name and ID
override ?
      if ( !(options & kNoNameAndIdOverride) &&  resolve fn. )
      {                                                // check for
supression
         if ( OC_WidgetIDtoAncestorSpec(0,aScopeID,&ancestorSpec) )
         {
            if ( !DB_OverridesForTypeAndItemDisabled(aDataType,...) )
               idOverrideOK = OC_CallSymbolicFunction(nameWithID, ...);
         }
      }
      if ( !idOverrideOK )
      {                                                // no name and ID
override...
         if ( !(options & kNoNameOverride) && resolve fullName )
         {                                             // discard the ID
part
            if ( OC_CallSymbolicFunction(fullName,iR,...) )
               return (ET_ViewHdl)~0;
         } else if ( aDataType )
         {
            dT = aDataType;
            vIf = DB_DoesInvokerExist(dT,actionName);
            if ( !vIf )
            {
               tid = TM_KeyTypeToTypeID(dT,NULL);
               if ( tid )                              // try climbing for
ancestors
                  tid = TM_GetParentTypeID(NULL,tid);
               if ( tid )
               {
                  aDataType = TM_GetTypeKeyType(NULL,tid);
                  if ( aDataType)
                  {
                     dp = DB_ResolveDataType(aDataType,NO);
                     while ( !dp )                     // up again!
                     {
                        tid = TM_KeyTypeToTypeID(aDataType,NULL);
                        if ( tid )
                           tid = TM_GetParentTypeID(NULL,tid);
                        if ( !tid )
                           return NULL;
                        aDataType = TM_GetTypeKeyType(NULL,tid);
                        dp = DB_ResolveDataType(aDataType,NO);
                     }
                     if ( dp )
                        stillLoop = YES;               // climb up and try
again...
                  }
               }
            } else
               return (vIf)(iR);
         }
      } else
         return (ET_ViewHdl)~0;
   }
   return NULL;
}
```

Hyperlinks

Given the type dependant, threaded invocation methodology described above, the next step is to implement the user-centric hyperlink capability. As an initial matter, the present invention uses a flexible dictionary system that can be used to build up lists of hyperlink targets and to rapidly look up the information necessary to invoke those targets when clicked on. The lexical analysis capability described in the Lexical Patent is the preferred system used to implement such a flexible dictionary system. Again, other lexical analyzer or dictionary system could also be used. In the context of hyperlinking, these dictionaries, which are implemented as lexical analyzer DBs, will be referred to as hyperlink domains. Given the lexical analyzer capabilities, adding an item to a domain (as in DB_AddToDomainDictionary) can be achieved by calling LX_Add() with the token string being the name involved and the token number being the corresponding unique ID. Persistence of these domains can be achieved by loading and saving the domain recognizer to/from a file placed within a hierarchical directory tree whose structure matches that of the underlying system ontology. Furthermore, looking up hyperlinks (as in DB_IsHyperlinkTarget) can be achieved by making a call to LX_Lex() (or a corresponding functional call). In the preferred embodiment, hyperlink domains can also be placed into active/inactive status. This can be most easily achieved by loading the corresponding lexical DBs into a linked list of such recognizers in memory on the local machine. The implementation of all hyperlink routines in the API below uses these calls to perform the functions described below.

The final component used by the present invention to support dynamic hyperlinks is a GUI framework that supports a multi-styled text display component. In other words, the hyperlink code (see PU_NotifyHyperlinkChange) implemented by the user environment must be able to examine the text in a control, and should a hyperlink phrase be found, must be able to alter the style of that portion of the text so that it is displayed appropriately for a hyperlink in the UI. This capability is supported by most non-trivial GUI frameworks (such as internet browsers) and is well-known to those skilled in the art. By combining a a framework that permits alteration of text styles to indicate hyperlinks and in which the environment supplied calls DB_Invoke() (which is tied to a system ontology) whenever the user clicks on any text that has been altered in this manner, we have a complete user-centric type and scope dependant hyperlink system.

API definitions

The API descriptions that follow give a sample embodiment of one basic public API that could be used by the present invention. This API is intended to be illustrative of the kinds of calls required and is by not intended to set forth any required implementation or otherwise exhaust the possible implementations. An API listing is also provided in Appendix A.

In the preferred embodiment, the function **OC_RegisterFunction()** registers a function by symbolic name for a given scope, so that it can be invoked from any other widget within that scope. The primary use of this functionality is to create a hyperlink registry to allow widgets to jump to other named locations without having to actually know where the location is or what the function it is calling actually does. In the preferred embodiment, the function registry is hierarchical with a registry potentially being attached to every ancestral level of the widget (including the widget itself). In this manner, it is possible to override the meaning of a function ("whoKnowsWhat") for an individual widget, a compound widget, a view, a view pack, or globally for the environment. This provides a great deal of flexibility in defining links between widgets and also allows certain functions to be overridden locally so that code that uses them can be modified without modifying the code itself. Preferably, functions specified as 'kFarFunction' are actually called in the context of the widget that registered them, not in that of the caller. On the other hand, 'near' functions are called in the context of the widget that makes the OC_CallSymbolicFunction() call. A typical symbolic function prototype might appear as follows:

```
void mySymbolicFunc          (                     // Symbolic function
              charPtr               aFuncName,     // I:Symbolic function name
              void                  *aParameter,   // IO:Parameter/Reply area (or
NULL)
              int32                 widgetID,      // I:Widget ID of caller
              int32                 options      // I:Various logical options
                             )                     // R:void
```

Preferably, any widget registering a function will de-register it at the functions terminate entry point. Otherwise, there is the possibility that the function may be called after the widget itself is dead. In the preferred embodment, a routine, such as OC_DeRegisterAllFuncs(), can be called to deregister any and all functions registered by a given widget regardless of the scope for which they were registered. An ancestorSpec of 'kViewPackSCOPE' is equivalent to 'kLocalViewSCOPE' if the calling widget is not within a view pack. When writing a 'kNearFunction' function, the near functions are called in the context of the widget that makes the OC_CallSymbolicFunction() call. In general the data associated with the installing widget may not be reliable and is it not safe to assume anything about the calling widget unless what the function requires/assumes in the 'aFuncDesc' parameter passed to this function is clearly described. A set of options, such as the 'kDistinguishFuncPtrs' options, can be used to allow multiple registrations of a given function name within the same widget but using distinct function addresses. Alternatively, only a single function 'funcName' can be registered for any given widget. For low-level libraries, when registering global type functions (e.g., "LanguageChange"), it is often helpful to distinguish registrations by different libraries.

In the preferred embodiment, the function **OC_DeRegisterFunction()**, can be used to remove a registered function from the function registry for the scope specified. If the function was not found at the specified scope, this function returns FALSE (and preferably does not log an error).

In the preferred embodiment, the function **OC_DisableFunction()** can be used to disable a registered function from the function registry for the scope specified. If the function was not found at the specified scope, this function returns FALSE (and does not log an error). Once disabled, the function will not be called until a corresponding OC_EnableFunction() call is made (for the same scope but not necessarily by the same widget). In the preferred embodiment, the function **OC_EnableFunction()** can be used to enable a registered function from in function registry for the scope specified if it has been previously disabled by a call to OC_DisableFunction(). If the function was not found at the specified scope, this function returns FALSE (and does not log an error). Since functions can be enabled and disabled by any widget within the scope, this mechanism serves as a convenient means of controlling function calls without having to add logic to the caller. In the preferred embodiment, the function **OC_FunctionIsDisabled()** allows you to determine is a specified function has been disabled for the selected scope. Similar functions could also be provided that enable or disable a function based on other factors, such as the time of day or date.

In the preferred embodiment, the function **OC_DeRegisterAllFuncs()** can be use d to remove all functions registered by the current widget (at any scope) from the function registry. If functions are removed successfully, TRUE is returned, otherwise FALSE is returned.

In the preferred embodiment, the function **OC_CallSymbolicFunction()** can be used to call a symbolic function from the symbolic function registry. Note that the result of this call reflects only whether the specified function could be found, not the result of actually calling it. In order to obtain a result back from a symbolic function (near or far), the address of a reply function (of type ET_SymbolicReply) must be provided which will be called in the same widget context as the OC_CallSymbolicFunction() call, and will be passed the 'aFuncParameter' value originally supplied (and also passed to the symbolic function). The parameter, if used, would be a pointer to a heap allocated block in the preferred embodiment. This approach allows the symbolic function to modify the value at that address, and allows the reply function (if specified) to examine the modified location to determine the result and then take whatever additional steps are necessary in the context of the original caller. In the preferred embodiment, the wrapping code possesses, dispossesses, and deletes the allocation (if used) according to the following rules:

1) If 'aReplyFunc' is specified, the allocation will be disposed of using KILL_PTR() after the reply function has been invoked.

2) If 'aReplyFunc' is not specified, the allocation will be disposed of using KILL_PTR() after the symbolic function has been invoked in the context of the registering widget for a 'far' function, or the calling widget for a 'near' function.

Far symbolic functions are actually called from within the event loop of the registering widget so those functions are responsible for causing the main loop of the widget to react (if required) either by posting an event/message, or other in-widget communications mechanisms. In particular, if the symbolic function needs to do something which might potentially cause the widget to be re-scheduled (such as UI operations or communication), it should preferably cause this to occur in the main widget loop, not do it itself.

Near symbolic functions are called immediately in the callers context and unlike far functions do not return to the caller until the function, and if specified, the reply function, have both been executed. If multiple different widgets have registered for the same symbolic function name at the effective scope, then every widget/function will be called (near and/or far) in sucession when 'aMatchWidgetID' is 0. This approach would permit broadcast type operations, for example. In the preferred embodiment, if any registration under the same name has occurred with a tighter scope, then the widget having the tighter scope will be called thereby suppressing all calls at the looser scope.

When multiple calls are made in this manner, all called functions share the identical 'aFuncParameter' storage, which is disposed when the last invoked function/reply completes. In the preferred embodiment, a number of options bits are reserved to allow the type of parameter passed in 'aFuncParameter' to be specified in those cases where a function accepts multiple parameter types. These definitions preferably have a one-for-one correspondence with the data type definitions for the options word. Some of the pareameters that could be used include:

kSymbParamTypeInvRec -- parameter is an ET_DBInvokeRecPtr
kSymbParamTypeInteger -- parameter is a pointer to a long
kSymbParamTypeString -- parameter is a C string pointer

In one embodiment, the 'kNoParameterDelete' supresses all possession, dispossession, and deletion of the 'aReplyFunc' value. This may be appropriate if the memory is to stay permanently owned by one widget, or if 'aFuncParameter' does not actually represent a heap pointer.

In the preferred embodiment, the function **OC_CountSymbolicFunctions()** can be used to count the number of widgets that are registered for a given symbolic function name at the effective scope. There are certain applications of symbolic functions that operate as a broadcast mechanism whereby multiple widgets register for a given symbolic function at a specified scope and all are called/invoked when the the OC_CallSymbolicFunction() call occurs. In most cases, the caller does not care how many functions are actually being triggered. In the event that it does, however, it may count the number and use the widget ID array returned by this function to pass to the 'matchWidgetID' parameter of other functions in order to select just a single instance (rather than all or just the first depending on the implementation). The number of widgets registered for a function at an effective scope is returned. In the preferred embodument, to specify a search of the global registry only, use '*aWidgetID' = kGlobalSCOPE on entry. '*aScopeID' (if specified) will be 0 on exit if the function was found in the global registry. The caller will dispose of the array returned in 'widgetIDs' when no longer required.

In the preferred embodiment, the function **OC_ResolveSymbolicFunction()** can be used to to determine if a given symbolic function exists, and if it does, the address of the function. The widget itself would not normally call the function (except by using OC_CallSymbolicFunction()) because many such functions are designed to be called in the context of the widget that registered them and fail if called from elsewhere. If the function pointer is not returned, then the function will return NULL. In this embodiment, to specify a search of the global registry only, use '*aWidgetID' = kGlobalSCOPE on entry. '*aScopeID' (if specified) will be 0 on exit if the function was found in the global registry.

In the preferred embodiment, the function **OC_SetSymbolicFuncData()**, can be called to attach data (or information) of a specified type to a registered symbolic function. A typical use of this function would be to attach an icon or picture to a function so that any function that is going to invoke the symbolic function can display the icon or picture associated with the function/destination. There are many other uses of this capability including communicating through the content of the data handle. The primary purpose of the ability for a sufficiently smart 'caller', however, is to establish certain information about the 'callee' before the call is made. If data is allocated and attached to a registered function, it must be deallocated at the time the function is de-registered. If an attempt is made to set function data from a widget other than the one that registered the function, it will fail. If operation is successful (meaning the registered widget was able to set function data), 0 is returned, otherwise an error number is returned.

In the preferred embodiment, the function **OC_GetSymbolicFuncData()** can be used to obtain the data (and its type) attached to a registered symbolic function. This information is associated with the function by the widget that registered it using OC_SetSymbolicFuncData(). The purpose of this data is to allow callers to obtain additional information about the function, without actually having to call it. If the 'aDataHandle' and 'aDataType' values come back as zero, there is no data associated with the function. Error numbers are preferably returned in the case of failure. The handle returned belongs to the widget that registered the symbolic function so any caller would preferably not de-allocate it or modify the contents (unless that is it's purpose).

In the preferred embodiment, the function **OC_SetSymbolicFuncFlags()** can be called to set the flags word associated with a symbolic function. Unlike the data associated with a symbolic function, the flags word can be altered by any widget within the scope. When setting the flags, it may be helpful to get the current flag settings using OC_GetSymbolicFuncFlags(), alter only those bits of interest, then set the flags using OC_SetSymbolicFuncFlags(). Failure to follow this protocol may result in confusion in cases where multiple widgets are manipulating the flags. In the preferred embodiment, the function **OC_GetSymbolicFuncFlags()** obtains the flags word associated with a registered symbolic function. This information is associated with the function by the widget that registered it using OC_SetSymbolicFuncFlags(). The purpose of this data is to allow callers to obtain additional information about the function, without actually having to call it.

In the preferred embodiment, the function **OC_GetSymbolicFuncDesc()** can be used to obtain the descriptive text (if any) associated with a registered symbolic function. If no description was supplied, the returned string contains "???". If descriptive text is not found, NULL is returned. In all other cases, a descriptive text handle is returned. The caller should dispose of the handle returned when no longer required.

In the preferred embodiment, the function **OC_ListSymbolicFunctions()** can be used to return an alpabetized, <CR> seperated list of all registered symbolic function names for the specified scope. preferably, the entries in the list have the format "www functionName" where 'www' is the widget ID of the widget that registered the function. To obtain the function description, the function OC_GetSymbolicFuncDesc() can be called and passed the 'www' and 'functionName' values. This function would returns a function list, or NULL if the list is empty. The caller should dispose of the handle returned when no longer required.

In the preferred embodiment, the function **OC_WidgetIDtoAncestorSpec()** can be used to convert a widget ID to the corresponding ancestor spec. If the widget ID is not ancestral to the calling widget, the function returns FALSE. In the preferred embodiment, the function **OC_AncestorSpecToWidgetID()** can be called to return the widget pointer corresponding to the ancestor specified relative to a given widget ID. The symbolic function registry uses this type of ancestor specification. In the preferred embodiment, the function **OC_LowestCommonAncestor()** returns the widget ID for the lowest common ancestor of the two widget IDs supplied (if it exists).

In the preferred embodiment, the function **DB_DefineHyperlinkDomain()** allows a hyperlink domain to be defined. The automatic hyperlinking facility assumes that hyperlink targets can be broken down first by data type (see DB_DefineDataType) and then within a given data type (People for example), as a set of groups or domains where each domain has a 'dictionary' (which is actually a lexical analyzer DB - see LX_MakeDB in the Lexical Patent incorporated herein) which contains a list of all target members that fall into that domain. In the example of the data type 'people', possible domains might be things such as politicians, military personel, or company staff. It is permissible that a given target (or person) be a member of any number of domains, providing that the person is unique within any given domain, or if not unique, is referenced by a different name for each multiple occurence (e.g., 'F16' and 'Falcon' might refer to the same target). Domains may be either system domains, meaning that the domain is common to all users of the system and are maintained by the system administrator, or they may be user domains, meaning that the domains are unique to each user of the system. If multiple domains recognize a given target, the first one to fire (which will be the last one to be activated) takes precedence regardless of the system or user attribute. Firing order can be controlled, if desired, by ensuring the preferred domain is activated after that of the domain over which it is preferred. In general, active system domains are loaded before user domains during startup, which normally has the effect of giving user domains precedence over system domains. Again, however, this precedence can be altered as desired. The effect of a hyperlink click is to invoke the "hyperlinkAction" action (the default if none is specified is "Display") for the data type of the domain which recognized the target. This means that hyperlinking is subject to all the same overriding and redirecting behaviors available via the DB_Invoke() function. This is useful because hyperlinks can be locally redirected when appropriate (with nested scope) while still following the default link if no override is found.

Once defined, a domain preferably becomes permanently known due to the fact that a domain dictionary file is created in the appropriate folder. The way to remove a domain is to call DB_UnDefineHyperlinkDomain(). Defining a domain that is already known or for which a domain dictionary file already exists, has no effect (this function returns TRUE with no action). Domains may also be organized into hierarchies by specifying the hierarchy path as a series of ancestral domains separated by colons (e.g., "animals:mammals:people"). This feature allows whole sub-trees to be activated or de-activated at once and allows flexibility in organizing domains according to any desired breakdown. Since a folder hierarchy is created to reflect the domain specification, it is important to ensure that all fields of a domain name meet the naming criteria for the underlying file system. In the preferred embodiment, all necessary ancestral folders will be created automatically when the domain is defined so it is not necessary to explicitly create the tree in a top down manner. To avoid confusion, domain names should be unique. Furthermore, it is not desirable to define a system and user domain name of the same name, nor is it desirable to have a domain name of a different 'aDataType' with the same name.

In the preferred embodiment, the function **DB_AddToDomainDictionary()** can be used to add a new target to the specified active hyperlink domain dictionary, thereby making it available as a hyperlink destination. To add targets to an inactive domain, it is best to temporarily activate (but not compact) the domain first. The most efficient way to add a series of targets to a given domain is to first ensure the domain is active (and not compacted), then add the targets (specifying the 'kNoSaveDomainToFile' option), and finally save the domain by making a call without the 'kNoSaveDomainToFile' option and NULL specified for 'aTargetName'. Lastly, the domain should be deactivated if it was not originally active. Preferably, this logic is handled automatically within a domain populator function as called via DB_CallDomainPopulator(). For correct operation, hyperlink targets MUST start with an alphanumeric character, not a delimiter or white-space. Alphanumeric characters may be in an alternate language as well as English so hyperlinks can operate in any language or script system.

In the preferred embodiment, the function **DB_SubFromDomainDictionary()** can be used to remove a target from the specified active hyperlink domain dictionary, thereby making it unavailable as a hyperlink destination. To remove targets from an inactive domain, the domain should be temporarily activated (but not compacted) first. If a series of targets to a given domain will be removed, the domain should be activated (or ensure the domain is active and not compacted), then calls made to remove the targets (specifying the 'kNoSaveDomainToFile' option), and the domain saved by making a call without the 'kNoSaveDomainToFile' option and NULL specified for 'aTargetName'. Lastly, the domain should be de-activate if it was not originally active.

In the preferred embodiment, the function **DB_NotifyHyperlinkChange()** should be called whenever some kind of change is made to the hyperlink dictionaries that requires the UI to be refreshed in order to determine again which hyperlinks are available. In the preferred use of this hyperlink API, this function does not need to be explicitly called since the calls are made automatically as appropriate.

In the preferred embodiment, the function **DB_IsHyperlinkTarget()** can be used to determine if a given string is a hyperlink target and, if so, what the data type, domain name, action, and unique ID are for that target. This function may be used to perform different hyperlinks using DB_Invoke() while specifying additional options or parameters based on detailed knowledge of the target, domain, or data type involved. Normally, DB_HyperlinkToTarget() would be used to explicitly invoke a hyperlink via some mechanism other than the automatic hyperlinking behavior provided for all text controls in the system. By using this function (followed by a call to DB_Invoke or DB_HyperlinkToTarget), it is possible to hyperlink to targets that are not in active domains. On input, if 'aDataType' is NULL or non-NULL with a value of zero, this is taken to imply that any key data type is acceptable, otherwise the value of '*aDataType' is used to restrict the search to only those active domains of the data type specified. On output, if 'aDataType' is non-NULL, it will hold the value of the key data type for which the target was found, or zero if not found. Additionally, on input, if 'aDomainName' is NULL, or non-NULL with a string value of "", this is taken to imply that any active domain name is acceptable, otherwise the value of the string pointed to by '*aDomainName' is taken to be a domain name in/below in which to look to the exclusion of all others. On output, if 'aDomainName' is non-NULL, the contents of the buffer to which the parameter value points will be replaced by the domain name in which the target was found (or an empty string if not found). Note that 'aDomainName' may be a partial path in which case the search for targets is restricted to all active domains below that path. In this embodiment, if and only if 'aDataType' and 'aDomainName' are specified explicitly, inactive domains will also be examined using this function. In all other cases, only active domains are considered. Because the contents of 'numChars' is set to the actual number of characters consumed when scanning for the target (found or otherwise), the string pointed to by 'aTargetName' can be an arbitrarily long sequence of text which is scanned for possible targets by successive calls. This is exactly what the function DB_FindNextHyperlinkInText() does. In such a case, the end of the string being scanned can be detected by the fact that 'numChars' will be zero. When skipping over characters, this function can also use a multilingual call to determine where alphanumeric strings begin and end. This means that hyperlinks can be either in English or the alternate language. It also means that when making a series of calls for a larger string, any trailing white-space and delimiters will be skipped such that only string elements that start with an alphanumeric character and are preceded by either a delimiter or white-space will actually be examined as potential targets. By making this simplification, the process of scanning a large block of text is greatly simplified and significantly optimized for speed. For this reason, hyperlink target name strings would preferably not begin with white-space or delimiters. Note that if 'maxChar' is specified (rather than defaulting it to zero), this routine will continue to scan until it reaches the 'maxChar' character position. This means that the text string supplied may contain embedded nulls.

In the preferred embodiment, the function **DB_HyperlinkToTarget()** can be used to find a hyperlink to the specified target. Since hyperlink handling is automatically supported for any and all text controls within the system, this function would only be used to invoke a hyperlink jump by some other mechanism. If data type and domain name are both specified explicitly, this function could also be used to hyperlink to a target that is not in an active domain (although this may be slower than a call for an active domain due to the need to temporarily load the domain dictionary).

In the preferred embodiment, the function **DB_IsKnownDomain()** can be used to determine if the specified domain is known or not. A domain is known if the domain dictionary file for the domain exists (even if the dictionary is empty). A domain does not have to be active to be known, however, the corresponding data type would preferably be defined. For a non-leaf domain, the value of 'isAutoActivate' will always be FALSE.

In the preferred embodiment, the function **DB_IsActiveDomain()** can be used to determine if the specified domain is active or not. Inactive domains are not automatically used when looking for targets.

In the preferred embodiment, the function **DB_ActivateDomain()** can be used to activate the specified domain. Activating a domain causes the domain dictionary to be loaded into memory and to be used automatically whenever any text within a text control is scanned for potential hyperlinks. In other words, all targets in the domain become potential hyperlinks. If the domain dictionary is compacted when it is activated, the dictionary will occupy significantly less memory. It is preferably not to add or remove targets from a compacted dictionary. A non-leaf domain may also be specified (domain name path ends in ':') in which case all leaf domains within (to any level) will be activated. In the preferred embodiment, the function **DB_DeActivateDomain()** can be used to deactivate a specified domain. Deactivating a domain causes the domain dictionary to be removed from memory thus preventing any targets within the domain from being used as automatic hyperlinks. If a domain has been designated in the optional hyperlinking administration window as 'auto activate' then deactivating it will have only a momentary effect since it will be re-activated almost immediately as a result of the auto-activation process.

In the preferred embodiment, the function **DB_GetDomainAction()** can be used to return the invoker action associated with the specified hyperlink domain. This action is used when calling DB_Invoke() during the hyperlinking process. The specified domain need not be active to discover its action.

In the preferred embodiment, the function **DB_SetDomainAutoFlags()** can be used to control wether the specified hyperlink domain is auto-activated during environment initialization. By designating a domain as auto-activating, all hyperlinks in that domain will be immediately available as soon as the application runs. For such domains, the 'autoCompact' flag can also be used to determine if the domain should be compacted when it is auto-activated.

In the preferred embodiment, the function **DB_SpecifyDomainPopulator()** can be used to specify a domain populator function to be used to fill out the dictionary associated with a domain. It is often the case that hyperlink domains correspond to entries in an external database of some kind. In the preferred embodiment, a populator function would perform a query(s) on that database to obtain the set of all targets in the domain and then loop adding the targets to the domain using DB_AddToDomainDictionary(). The hyperlink configuration view allows the invocation of the populator function for any given domain as well as configuration of which domains are to be active at any given time. At the time the domain populator is called, the domain itself will preferably have been made active (temporarily if appropriate) and the domain dictionary in memory will be empty. If the domain populator function returns FALSE, the domain dictionary in memory will be discarded and replaced (if appropriate) with the dictionary from the domain dictionary file. During all calls from within a domain populator function, the save to file behavior of DB_AddToDomainDictionary() is automatically inhibited for this reason. A typical domain populator function might appear as follows:

```
EngErr myDomainPopulator    (                              // my domain populator
            ET_TypeID           aTypeID,                   // I:Data type for the domain
            charPtr             aDomainName,               // I:Domain name
            charPtr             populatorDescription,// I:Populator description
            long                aParam                     // I: custom parameter or 0
                            )                              // R:0 for success,else error #
```

In the preferred embodiment, the function **DB_CallDomainPopulator()** can be used to call the hyperlink domain populator function (if there is one), passing an arbitrary parameter. When populator functions are called from within the standard hyperlink configuration UI, this parameter will be zero.

In the preferred embodiment, the function **DB_UseDefaultDomainPopulator()** can be used to specify the use of the generic hyperlink domain populator provided for persistent data types derived from the key type 'DTUM' (i.e., Datum).

In the preferred embodiment, the function **DB_FindNextHyperlinkInText()** can be used to scan a block of text looking for hyperlink targets within it. In the preferred embodiment, the function is called with both 'aDataType' and 'aDomainName' set to zero, which causes it to utilize all active hyperlink domain dictionaries to scan the text looking for a match. The data type may be restricted or a partial hyperlink domain specified. In particular, if the data type is specified and a full or partial domain name is given, this function will also find targets in any inactive hyperlink domains specified. See DB_IsHyperlinkTarget() for details on restricting the hyperlink search. This function forms the basis of the automatic hyperlinking capability provided by the UI encapsulation layer whereby all text in a text control is scanned and hyperlinks inserted (by turning the target word/phrase blue and underlining it, for example) and handled when clicked on by the UI layer. This function will return successive hyperlinks on each call until there are no more hyperlinks left in the text at which time it will return FALSE. The value of '*context' should be set to zero to start the scanning process, otherwise the value should be preserved between successive calls to this function.

In the preferred embodiment, the function **DB_ListKnownDomains()** can be used to return a hierarchical Lex DB containing all known system or user hyperlink domains. The resulting Lex DB may be used either to recognize domain names, or it may be used to process/list the domains using the facilities provided by LX_List() and the associated functions such as LX_PruneList() and LX_Save/RestoreListContext(). The LexDB returned by this function includes the data type name prefix in the domain paths. Calls to other functions in this API do not contain this prefix for the 'aDomainName' parameter.

In the preferred embodiment, the function **DB_ListActions()** can be used to return an alphabetized, carriage return (<nl>) separated list of all the invoker actions supported for a given key data type. The list is repeatedly initialized until the tables are exhausted at which time the next symbol is listed and displayed. NULL is returned in case of an error. The list of actions returned may include actions for which there is not actually an invoker function (see DB_DefineInvoker) but for which symbolic overrides have been defined. The routine DB_DoesInvokerExist() can be used to determine if this is the case.

In the preferred embodiment, the function **DB_DataTypeToName()** can be used to return the full symbolic name of the specified key data type. In the preferred embodiment, the function **DB_NameToDataType()** returns the key data type given a full symbolic name, type name, or an alternate name. In the preferred embodiment, the function **DB_OSTypeToString()** can be used to convert a long to display as a character string. The normal application would be for use with OSTypes.

In the preferred embodiment, the function **DB_OverrideForTypeAndItemExists()** can be used to determine if an override exists for the specified key data type and item ID and, if so, the scope relative to the asking widget. This information can be used to determine if it is possible to display a given type within a particular calling context.

In the preferred embodiment, the function **DB_OverrideForTypeAndItem()** can be called in order to register to handle a given action for a specified key data type and a unique ID of that type. This capability can be used to cause re-mapping of the view invoked on a DB_Invoke() call for an desired scope. This is particularly useful in ensuring that if data for a given item is already being displayed, another view is not launched but instead the existing view is simply brought forward. All items of a given type can be re-directed using DB_OverrideForType(). In the preferred embodiment, the function DB_Invoke() will first check for a specific override and then for a general one. In the preferred embodiment, the function **DB_UndoOverrideForTypeAndItem()** can be used to remove an override registered using DB_OverrideForTypeAndItem(). If no such override exists, the function will do nothing.

In the preferred embodiment, the function **DB_DisableOverrideForTypeAndItem()** can be used to supress overrides for a given key data type, ID, and scope. The suppression remains in effect until a call to DB_EnableOverrideForTypeAndItem() is made. Any widget may remove the suppression, not just the one registering it. When called with 'anItemID' of zero, this function disables all ID based overrides for the type and scope. This disable is in addition to any ID specific disables that may be in effect, and can be removed by passing 'anItemID' of zero to DB_EnableOverrideForTypeAndItem(). In the preferred embodiment, the function **DB_EnableOverrideForTypeAndItem()** can be used to remove any supression for a given type, ID, and scope registered by DB_DisableOverrideForTypeAndItem().

In the preferred embodiment, the function **DB_OverrideForType()** can be called in order to register to handle a given action for a specified key data type. This capability can be used to cause re-mapping of the view invoked on a DB_Invoke() call for an desired scope. Note that you can re-direct specific items of a given key data type using DB_OverrideForTypeAndItem(). In the preferred embodiment, the function DB_Invoke() will first check for a specific override and then for a general one. In the preferred embodiment, the function **DB_UndoOverrideForType()** removes an override registered using DB_OverrideForType(). If no such override exists, the function will do nothing.

In the preferred embodiment, the function **DB_DisableOverrideForType()** can be used to supress overrides for a given type and scope. The suppression remains in effect until a call to DB_EnableOverrideForType() is made. Any widget may remove the suppression, not just the one registering it. In the preferred embodiment, the function **DB_EnableOverrideForType()** may be called to remove any suppression for a given key data type and scope registered by DB_DisableOverrideForType(). In the preferred embodiment, the function **DB_OverridesForTypeDisabled()** can be called to determine if overrides for a given key data type and scope have been suppressed.

In the preferred embodiment, the function **DB_OverridesForTypeAndItemDisabled()** can be used to determine if overrides for a given key data type,ID and scope have been suppressed.

In the preferred embodiment, the function **DB_OverrideForTypeExists()** can be used to determine if an override exists for the specified key data type, and if so with what scope relative to the asking widget. This information can be used to determine if it is possible to display a given type within a particular calling context. Even though an override exists, it may have been disabled. Preferably, DB_OverridesForTypeAndItemDisabled() is used to determine if this is the case.

In the preferred embodiment, the function **DB_DefineInvoker()** can be used to define the view invoker function that should be called when an attempt is made to perform a specified invoker action on a given key data type. For example, the 'actionName' parameter might be "Display", in which case any subsequent call to DB_Invoke() for the action "Display" will result in the specified invoker function being called. The invoker function is responsible to instantiating or launching the view necessary to perform the requested action for the specified data type. Custom named invoker actions may be defined for each different data type as appropriate. In the preferred embodiment, certain predefined action types are defined and would preferably be supported by a given key data type (by defining the necessary invokers) wherever possible:

"Display" The invoker should display the selected data item as appropriate, but may not allow editing. This action is required for IP notification to be effective in this embodiment.

"Edit" The invoker should display and allow edit/update of the data item.

"Select" The invoker should display a list of items and notify the caller of any selection made by the user.

"Print" The invoker should print the selected item in the appropriate format (may not be a view launch).

"Info" Display information associated with the type (in a widget modal window, NOT actually a view launch). This action is required to place a "Show Info" button in the pending views window for this type in this embodiment.

If 'anInvokerFn' is NULL, this function can be used to define an action type to Database such that the available actions for the type can be returned on a subsequent DB_ListActions() call or used in action overrides. Whenever an override is registered for a defined type (i.e., from a call to DB_DefineDataType), the corresponding action is automatically registered for the type using this function. In this way, it is possible to determine the full set of actions (whether invoker based or via symbolic overrides) for a type using DB_ListActions(). Any type manager type that is descended from a type manager type that is also a key data type will inherit the invokers and actions of the key type. In the preferred embodiment, the function **DB_UnDefineInvoker()** can be used to remove the existing definition of an invoker function for the specified key data type and action, presumably in preparation for defining a replacement function using DB_DefineInvoker(). If invoker is removed, TRUE is returned, otherwise FALSE is returned. In the preferred embodiment, the function **DB_DoesInvokerExist()** can be used to determine if an invoker function exists for the specified key data type and action. An invoker function address is returned, if it exists; otherwise NULL.

In the preferred embodiment, the function **DB_Invoke()** can be used to call the registered invoker function for the key data type and action specified. The result is normally to instantiate or launch another view. It is also possible, however, that the function will execute entirely within the original caller's widget context. Examples of such invokers might be "Print" or "Info". This function, and the 'ET_DBInvokeRec' record that it uses, could also be used for other launcher/launchee situations even if the implementation below varies. In all cases, the 'anItemID' field of 'iR' would preferably be filled out with a unique item number that can be used by the invoked function to determine which item of a set of items is required. The caller, depending on the situation and depending upon whether the caller has already fetched the information necessary to accomplish the invocation, may also fill out other fields. In order to provide sufficient flexibility to allow general use, this routine will preferably accept an 'aDataType' value of zero as meaning that there is no true data type corresponding to this invocation, but nonetheless the routine DB_Invoke() is being used. In this case, it is preferably that the 'dataItemType' field of the 'iR' record contains a string describing the data type involved (e.g., "My data type"). DB_Invoke() will take this string, concatenate it to the 'actionName' string (for example "DisplayMy data type"), and check for the presence of a registered symbolic function with that name (see OC_RegisterSymbolicFunction()). If such a function is found, it will be invoked.

Within this symbolic function, any action necessary to accomplish the actual invocation can be performed. The same symbolic function override capability exists for true data types, i.e., if a function "DisplayNewswire" exists for the data type whose name is

'Newswire' then it will be called in preference to the registered invoker function for 'Newswire'. This feature allows registeration of invoker overrides at various scopes in order to re-direct the behavior. This feature is also what allows DB_Invoke() to be used as a universal invocation method (see description above). In the preferred embodiment, the functions DB_OverrideForTypeAndItem() and DB_OverrideForType() are provided to allow a convenient means of overriding (using symbolic functions) the function invoked for either a specific item ID and data type (see DB_OverrideForTypeAndItem) or a specific data type regardless of ID (see DB_OverrideForType). In the described embodiment, the 'iR' parameter must be a pointer allocated in the heap, it cannot be a stack variable. If a result (or an error) is returned, the original caller is responsible for disposing of 'iR'. In the preferred embodiment, if the 'actionName' parameter is NULL, this function attempts to invoke the "Display" action (assuming an invoker for "Display" has been defined).

In the preferred embodiment, the function **PU_CursorToHyperlink()** can be called by the environment within the widget context during idle time. This function can be used to determine what hyperlink, if any, the cursor/mouse is currently over provided that it is called within the appropriate widget context. By doing this, the environment knows when a user clicks on a hyperlink within some text and can automatically invoke the link as necessary. In systems including drag-and-drop, this mechanism is extended to automatically follow any hyperlink over which the user hovers while executing a drag so that the user can use the hyperlink mechanism as part of the navigation process during drag-and-drop operations.

In the preferred embodiment, the function **PU_NotifyHyperlinkChange()** can be called automatically by the environment in order to ensure that all text controls display the correct hyperlinks within them (see DB_NotifyHyperlinkChange). In the preferred embodiment, the function scans all widget contexts, and all windows within those widgets looking for text controls. The function then examines the text within those controls for possible hyperlinks (see DB_FindNextHyperlinkInText) and if one is found, alters the style run for the text portion that represents the hyperlink to the appearance necessary to indicate to the user that a hyperlink is present. This means that any UI displayed by the system will always show whatever hyperlinks exist for the currently active domains and this appearance will be dynamically updated should any change occur in the users hyperlinking configuration. This feature enables a truly dynamic and "real time" hyperlinking system.

The foregoing description of the preferred embodiments of the invention has been presented for the purposes of illustration and description. For example, although described with respect to the C programming language, any programming language could be used to implement this invention. Additionally, the claimed system and method should not be limited to the particular API disclosed. The descriptions of the header structures should also not be limited to the embodiments described. While the sample pseudo code provides examples of the code that may be used, the plurality of implementations that could in fact be developed is nearly limitless. Finally, although described with reference to "Internet" terms such as hyperlinking, this invention could be applied to content from any number of different envionments. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

SYSTEM AND METHOD FOR ANALYZING DATA

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

Lexical analyzers are generally used to scan sequentially through a sequence or "stream" of characters that is received as input and returns a series of language tokens to the parser. A token is simply one of a small number of values that tells the parser what kind of language element was encountered next in the input stream. Some tokens have associated *semantic* values, such as the name of an identifier or the value of an integer. For example if the input stream was:

```
dst = src + dst->moveFrom
```

After passing through the lexical analyzer, the stream of tokens presented to the parser might be:

```
 (tok=1,     string="dst")  -- i.e., 1 is the token for identifier
(tok=100,  string="=")
(tok=1,string="src")
(tok=101,  string="+")
(tok=1,string="dst")
(tok=102,  string="->")
(tok=1,string="moveFrom")
```

To implement a lexical analyzer, one must first construct a Deterministic Finite Automaton (DFA) from the set of tokens to be recognized in the language. The DFA is a kind of state machine that tells the lexical analyzer given its current state and the current input character in the stream, what new state to move to. A finite state automaton is deterministic if it has no transitions on input Є (epsilon) and for each state, S, and symbol, A, there is at most one edge labeled A leaving S. In the present art, a DFA is constructed by first constructing a Non-deterministic Finite Automaton (NFA). Following construction of the NFA, the NFA is converted into a corresponding DFA. This process is covered in more detail in most books on compiler theory.

APPENDIX 1

In Figure 1, a state machine that has been programmed to scan all incoming text for any occurrence of the keywords “dog”, “cat”, and “camel” while passing all other words through unchanged is shown. The NFA begins at the initial state (0). If the next character in the stream is ‘d’, the state moves to 7, which is a non-accepting state. A non-accepting state is one in which only part of the token has been recognized while an accepting state represents the situation in which a complete token has been recognized. In Figure 1, accepting states are denoted by the double border. From state 7, if the next character is ‘o’, the state moves to 8. This process will then repeat for the next character in the stream. If the lexical analyzer is in an accepting state when either the next character in the stream does not match or in the event that the input stream terminates, then the token for that accepting state is returned. Note that since “cat” and “camel” both start with “ca”, the analyzer state is "shared" for both possible "Lexemes". By sharing the state in this manner, the lexical analyzer does not need to examine each complete string for a match against all possible tokens, thereby reducing the search space by roughly a factor of 26 (the number of letters in the alphabet) as each character of the input is processed. If at any point the next input token does not match any of the possible transitions from a given state, the analyzer should revert to state 10 which will accept any other word (represented by the dotted lines above). For example if the input word were “doctor”, the state would get to 8 and then there would be no valid transition for the ‘c’ character resulting in taking the dotted line path (i.e., any other character) to state 10. As will be noted from the definition above, this state machine is an NFA not a DFA. This is because from state 0, for the characters ‘c’ and ‘d’, there are two possible paths, one directly to state 10, and the others to the beginnings of “dog” and “cat”, thus we violate the requirement that there be one and only one transition for each state-character pair in a DFA.

Implementation of the state diagram set forth in Figure 1 in software would be very inefficient. This is in part because, for any non-trivial language, the analyzer table will need to be very large in order to accommodate all the "dotted line transitions". A standard algorithm, often called ‘subset construction’, is used to convert an NFA to a corresponding DFA. One of the problems with this algorithm is that, in the worst-case scenario, the number of states in the resulting DFA can be exponential to the number of NFA states. For these reasons, the ability to construct languages and parsers for complex languages on the fly is needed. Additionally, because lexical analysis is occurring so pervasively and often on many systems, lexical analyzer generation and operation needs to be more efficient.

SUMMARY OF INVENTION

The following system and method provides the ability to construct lexical analyzers on the fly in an efficient and pervasive manner. Rather than using a single DFA table and a single method for lexical analysis, the present invention splits the table describing the automata into two distinct tables and splits the lexical analyzer into two phases, one for each table. The two phases consist of a single transition algorithm and a range transition algorithm, both of which are table driven and, by eliminating the need for NFA to DFA conversion, permit the dynamic modification of those tables during operation. A third 'entry point' table may also be used to speed up the process of finding the first table element from state 0 for any given input character (i.e, states 1 and 7 in Figure 1). This third table is merely an optimization and is not essential to the algorithm. The two tables are referred to as the 'onecat' table and the 'catrange' tables. The onecat table includes records, of type "ET_onecat", that include a flag field, a catalyst field, and an offset field. The catalyst field of an ET_onecat record specifies the input stream character to which this record relates. The offset field contains the positive (possibly scaled) offset to the next record to be processed as part of recognizing the stream. Thus the 'state' of the lexical analyzer in this implementation is actually represented by the current 'onecat' table index. The 'catrange' table consists of an ordered series of records of type ET_CatRange, with each record having the fields 'lstat' (representing the lower bound of starting states), 'hstat' (representing the upper bound of starting states), 'lcat' (representing the lower bound of catalyst character), 'hcat' (representing the upper bound of catalyst character) and 'estat' (representing the ending state if the transition is made).

The method of the present invention begins when the analyzer first loops through the 'onecat' table until it reaches a record with a catalyst character of 0, at which time the 'offset' field holds the token number recognized. If this is not the final state after the loop, the lexical analyzer has failed to recognize a token using the 'onecat' table and must now re-process the input stream using the 'catrange' table. The lexical analyzer loops re-scanning the 'catrange' table from the beginning for each input character looking for a transition where the initial analyzer state lies between the 'lstat' and 'hstat' bounds, and the input character lies between the 'lcat' and 'hcat' bounds. If such a state is found, the analyzer moves to the new state specified by 'estat'. If the table runs out (denoted by a record with 'lstat' set to 255) or the input string runs out, the loop exits.

The invention also provides a built-in lexical analyzer generator to create the catrange and onecat tables. By using a two-table approach, the generation phase is extremely fast but more importantly, it can be incremental, meaning that new symbols can be added to the analyzer while it is running. This is a key difference over conventional approaches because it opens up the use of the lexical analyzer for a variety of other purposes that would not normally be possible. The two-phase approach of the present invention also provides significant advantages over standard techniques in terms of performance and flexibility when implemented in software, however, more interesting applications exist when one considers the possibility of a hardware implementation. As further described below, this invention may be implemented in hardware, software, or both.

BRIEF DESCRIPTION OF THE FIGURES

Figure 1 illustrates a sample non-deterministic finite automaton.

Figure 2 illustrates a sample ET_onecat record using the C programming language.

Figure 3 illustrates a sample ET_catrange record using the C programming language.

Figure 4 illustrates a state diagram representing a directory tree.

Figure 5 illustrates a sample structure for a recognizer DB.

Figure 6 illustrates a sample implementation of the Single Transition Module.

Figure 7 illustrates the operation of the Single Transition Module.

Figure 8 illustrates a logical representation of a Single Transition Module implementation.

Figure 9 illustrates a sample implementation of the Range Transition Module.

Figure 10 illustrates a complete hardware implementation of the Single Transition Module and the Range Transition Module.

DETAILED DESCRIPTION OF THE PREFERRED EMBODIMENT

The following description of the invention references various C programming code examples that are intended to clarify the operation of the method and system. This is not intended to limit the invention as any number of programming languages or implementations may be used.

The present invention provides an improved method and system for performing lexical analysis on a given stream of input. The present invention comprises two distinct tables that describe the automata and splits the lexical analyzer into two phases, one for each table. The two phases consist of a single transition algorithm and a range transition algorithm. A third 'entry point' table may also be used to speed up the process of finding the first table element from state 0 for any given input character (i.e, states 1 and 7 in Figure 1). This third table is merely an optimization and is not essential to the algorithm. The two tables are referred to as the 'onecat' table and the 'catrange' tables.

Referring now to Figure 2, programming code illustrating a sample ET_onecat record 200 is provided.The onecat table includes records, of type "ET_onecat", that include a flag field, a catalyst field, and an offset field. The catalyst field of an ET_onecat record specifies the input stream character to which this record relates. The offset field contains the positive (possibly scaled) offset to the next record to be processed as part of recognizing the stream. Thus the 'state' of the lexical analyzer in this implementation is actually represented by the current 'onecat' table index.The 'onecat' table is a true DFA and describes single character transitions via a series of records of type ET_onecat 200. A variety of specialized flag definitions exist for the flags field 210 but for the purposes of clarity, only 'kLexJump' and 'kNeedDelim' will be considered. The catalyst field 205 of an ET_onecat record 200 specifies the input stream character to which this record relates. The offset field 215 contains the positive (possibly scaled) offset to the next record to be processed as part of recognizing the stream. Thus the 'state' of the lexical analyzer in this implementation is actually represented by the current 'onecat' table index. For efficiency, the various 'onecat' records may be organized so that for any given starting state, all possible transition states are ordered alphabetically by catalyst character.

The basic algorithm for the first phase of the lexical analyzer, also called the onecat algorithm, is provided. The algorithm begins by looping through the 'onecat' table (not shown) until it reaches a record with a catalyst character of 0, at which time the 'offset' field 215 holds the token number recognized. If this is not the final state after the loop, the algorithm has failed to recognize a token using the 'onecat' table and the lexical analyzer must now re-process the input stream from the initial point using the 'catrange' table.

```
ch = *ptr;                              // 'ptr'
tbl = &onecat[entryPoint[ch]];          // initialize using 3rd table
for ( done = NO ;; )
{
tch = tbl->catalyst;
state = tbl->flags;
    if ( !*ptr ) done = YES;            // oops! the source string ran out!
if ( tch == ch )                        // if 'ch' matches catalyst char
{                                       //    match found, increment to next
if ( done ) break;                      // exit if past the terminating NULL
tbl++;                                  // increment pointer if char accepted
ptr++;                                  // in the input stream.
ch = *ptr;
}
else if ( tbl->flags & kLexJump )
tbl += tbl->offset;                     // there is a jump alternative available
else break;                             // no more records, terminate loop
}
match = !tch && (*ptr is a delimiter || !(state & (kNeedDelim+kLexJump)));
```

```
if ( match ) return tbl->offset;      // on success, offset field holds token#
```

Referring now to Figure 3, sample programming code for creating an ET_Catrange record 300 is shown. The 'catrange' table (not shown) consists of an ordered series of records of type ET_CatRange 300. In this implementation, records of type ET_CatRange 300 include the fields 'lstat' 305 (representing the lower bound of starting states), 'hstat' 310 (representing the upper bound of starting states), 'lcat' 315 (representing the lower bound of catalyst character), 'hcat' 320 (representing the upper bound of catalyst character) and 'estat' 325 (representing the ending state if the transition is made). These are the minimum fields required but, as described above, any number of additional fields or flags may be incorporated.

A sample code implementation of the second phase of the lexical analyzer algorithm, also called the catrange algorithm, is set forth below.

```
tab = tab1 = &catRange[0];
state = 0;
ch = *ptr;
for (;;)
{                                  // LSTAT byte = 255 ends table
if ( tab->lstat == 255 ) break;
else if ( ( tab->lstat <= state && state <= tab->hstat ) &&
( tab->lcat <= ch && ch <= tab->hcat ) )
{                               // state in range & input char a valid catalyst
state = tab->estat; // move to final state specified
ptr++;                  // accept character
ch = *ptr;
if ( !ch ) break;       // whoops! the input string ran out
tab = tab1;             // start again at beginning of table
}
else tab++;                 // move to next record if not end
}
if ( state > maxAccState || *ptr not a delimiter && *(ptr-1) not a delimiter )
return bad token error
return state
```

As the code above illustrates, the process begins by looping and re-scanning the 'catRange' table from the beginning for each input character looking for a transition where the initial analyzer state lies between the 'lstat' 305 and 'hstat' 310 bounds, and the input character lies between the 'lcat' 315 and 'hcat' 320 bounds. If such a state is found, the analyzer moves to the new state specified by 'estat' 325. If the table runs out (denoted by a record with 'lstat' set to 255) or the input string runs out, the loop exits. In the preferred embodiment, a small number of tokens will be handled by the 'catRange' table (such an numbers, identifiers, strings etc.) since the reserved words of the language to be tokenized will be tokenized by the 'onecat' phase. Thus, the lower state values (i.e. <64) could be reserved as accepting while states above that would be considered non-accepting. This boundary line is specified for a given analyzer by the value of 'maxAccState' (not shown).

To illustrate the approach, the table specification below is sufficient to recognize all required 'catRange' symbols for the C programming language:

```
0 1 1 a z          <eol> 1 = Identifier
0 1 1 _ _          <eol> more identifier
1 1 1 0 9          <eol> more identifier
0 0 100 ' '        <eol> ' begins character constant
100 100 101 \ \    <eol> a \ begins character escape sequence
```

101 102 102 0 7	<eol> numeric character escape sequence
101 101 103 x x	<eol> hexadecimal numeric character escape sequence
103 103 103 a f	<eol> more hexadecimal escape sequence
103 103 103 0 9	<eol> more hexadecimal escape sequence
100 100 2 ' '	<eol> ' terminates the character sequence
102 103 2 ' '	<eol> you can have multiple char constants
100 103 100	<eol> 2 = character constant
0 0 10 0 0	<eol> 10 = octal constant
10 10 10 0 7	<eol> more octal constant
0 0 3 1 9	<eol> 3 = decimal number
3 3 3 0 9	<eol> more decimal number
0 0 110 . .	<eol> start of fp number
3 3 4 . .	<eol> 4 = floating point number
10 10 4 . .	<eol> change octal constant to fp #
4 4 4 0 9	<eol> more fp number
110 110 4 . .	<eol> more fp number
3 4 111 e e	<eol> 5 = fp number with exponent
10 10 111 e e	<eol> change octal constant to fp #
111 111 5 0 9	<eol> more exponent
111 111 112 + +	<eol> more exponent
0 0 0 \ \	<eol> continuation that does not belong to anything
111 111 112 - -	<eol> more exponent
112 112 5 0 9	<eol> more exponent
5 5 5 0 9	<eol> more exponent
4 5 6 f f	<eol> 6 = fp number with optional float marker
4 5 6 1 l	<eol> more float marker
10 10 120 x x	<eol> beginning hex number
120 120 7 0 9	<eol> 7 = hexadecimal number
120 120 7 a f	<eol> more hexadecimal
7 7 7 0 9	<eol> more hexadecimal
7 7 7 a f	<eol> more hexadecimal
7 7 8 1 l	<eol> 8 = hex number with L or U specifier
7 7 8 u u	<eol>

3 3 9 l l	<eol> 9 = decimal number with L or U specifier
3 3 9 u u	<eol>
10 10 11 l l	<eol> 11 = octal constant with L or U specifier
10 10 11 u u	<eol>
0 0 130 " "	<eol> begin string constant...
130 130 12 " "	<eol> 12 = string constant
130 130 13 \ \	<eol> 13 = string const with line continuation '\'
13 13 131 0 7	<eol> numeric character escape sequence
131 131 131 0 7	<eol> numeric character escape sequence
13 13 132 x x	<eol> hexadecimal numeric character escape sequence
131 132 12 " "	<eol> end of string
13 13 130	<eol> anything else must be char or escape char
132 132 132 a f	<eol> more hexadecimal escape sequence
132 132 132 0 9	<eol> more hexadecimal escape sequence
130 132 130	<eol> anything else is part of the string

In this example, the 'catRange' algorithm would return token numbers 1 through 13 to signify recognition of various C language tokens. In the listing above (which is actually valid input to the associated lexical analyzer generator), the 3 fields correspond to the 'lstat' 305, 'hstat' 310, 'estat' 325, 'lcat' 315 and 'hcat' 320 fields of the ET_CatRange record 300. This is a very compact and efficient representation of what would otherwise be a huge number of transitions in a conventional DFA table. The use of ranges in both state and input character allow us to represent large numbers of transitions by a single table entry. The fact that the table is re-scanned from the beginning each time is important for ensuring that correct recognition occurs by arranging the table elements appropriately. By using this two pass approach, we have trivially implemented all the dotted-line transitions shown in the initial state machine diagram as well as eliminating the need to perform the NFA to DFA transformation. Additionally since the 'oneCat' table can ignore the possibility of multiple transitions, it can be optimized for speed to a level not attainable with the conventional NFA->DFA approach.

The present invention also provides a built-in lexical analyzer generator to create the tables described. 'CatRange' tables are specified in the format provided in Figure 3, while 'oneCat' tables may be specified via application programming interface or "API" calls or simply by specifying a series of lines of the form provided below.

```
[ token# ] tokenString [ . ]
```

As shown above, in the preferred embodiment, a first field is used to specify the token number to be returned if the symbol is recognized. This field is optional, however, and other default rules may be used. For example, if this field is omitted, the last token number + 1 may be used instead. The next field is the token string itself, which may be any sequence of characters including whitespace. Finally, if the trailing period is present, this indicates that the 'kNeedDelim' flag (the flags word bit for needs delimiter, as illustrated in Figure 2) is false, otherwise it is true.

Because of the two-table approach, this generation phase is extremely fast. More importantly, however, the two table approach can be incremental. That is, new symbols can be added to the analyzer while it is running. This is a key difference over conventional approaches because it opens up the use of the lexical analyzer for a variety of other purposes that would not normally be possible. For example, in many situations there is a need for a symbolic registration database wherein other programming code can register items identified by a unique 'name'. In the preferred embodiment, such registries are implemented by dynamically adding the symbol to a 'oneCat' table, and then using the token number to refer back to whatever was registered along with the symbol, normally via a pointer. The advantage of this approach is the speed with which both the insertion and the lookup can occur. Search time in the registry is also dramatically improved over standard searching techniques (e.g., binary search). Specifically, search time efficiency (the "Big O" efficiency) to lookup a given word is proportional to the log (base N) of the number of characters in the token, where 'N' is the number of different ASCII codes that exist in significant proportions in the input stream. This is considerably better than standard search techniques. Additionally, the trivial nature of the code needed to implement a lookup registry and the fact that no structure or code needs to be designed for insertion, removal and lookup, make this approach very convenient.

In addition to its use in connnection with flat registries, this invention may also be used to represent, lookup, and navigate through hierarchical data. For example, it may be desirable to 'flatten' a complete directory tree listing with all files within it for transmission to another machine. This could be easily accomplished by iterating through all files and directories in the tree and adding the full file path to the lexical analyzer database of the present invention. The output of such a process would be a table in which all entries in the table were unique and all entries would be automatically ordered and accessible as a hierarchy.

Referring now to Figure 4, a state diagram representing a directory tree is shown. The directory tree consists of a directory A containing sub-directories B and C and files F1 and F2 and sub-directory C contains F1 and F3. A function, LX_List(), is provided to allow alphabetized listing of all entries in the recognizer database. When called successively for the state diagram provided in Figure 6, it will produce the sequence:

"A:", "A:B:", "A:C:", "A:C:F1", "A:C:F3", "A:F1", "A:F2"

Furthermore, additional routines may be used to support arbitrary navigation of the tree. For example, routines could be provided that will prune the list (LX_PruneList()), to save the list (LX_SaveListContext()) and restore the list (LX_RestoreListContext()). The routine LX_PruneList() is used to "prune" the list when a recognizer database is being navigated or treated as a hierarchical data structure. In one embodiment, the routine LX_PruneList() consists of nothing more than decrementing the internal token size used during successive calls to LX_List(). The effect of a call to LX_PruneList() is to remove all descendant tokens of the currently listed token from the list sequence. To illustrate the point, assume that the contents of the recognizer DB represent the file/folder tree on a disk and that any token ending in ':' is a folder while those ending otherwise are files. A program could easily be developed to enumerate all files within the folder folder "Disk:MyFiles:" but not any files contained within lower level folders. For example, the following code demonstrates how the LX_PruneList() routine is used to "prune" any lower level folders as desired:

```
tokSize = 256;                                    // set max file path length
prefix = "Disk:MyFiles:";
toknum = LX_List(theDB,0,&tokSize,0,prefix);      // initialize to start folder path
while ( toknum != -1 )                            // repeat for all files
{
   toknum = LX_List(theDB,fName,&tokSize,0,prefix);  // list next file name
```

```
    if (toknum != -1 )                              // is it a file or a folder ?
        if ( fName[tokSize-1] == ':' )              // it is a folder
            LX_PruneList(theDB)                     // prune it and all it's children
        else                                        // it is a file...
            -- process the file somehow
}
```

In a similar manner, the routines LX_SaveListContext() and LX_RestoreListContext() may be used to save and restore the internal state of the listing process as manipulated by successive calls to LX_List() in order to permit nested/recursive calls to LX_List() as part of processing a hierarchy. These functions are also applicable to other non-recursive situations where a return to a previous position in the listing/navigation process is desired. Taking the recognizer DB of the prior example (which represents the file/folder tree on a disk), the folder tree processing files within each folder at every level could be recursively walked non-recursively by simply handling tokens containing partial folder paths. If a more direct approach is desired, the recursiveness could be simplified. The following code illustrates one direct and simple process for recursing a tree:

```
void myFunc ( charPtr folderPath )
{
    tokSize = 256;                                     // set max file path length
    toknum = LX_List(theDB,0,&tokSize,0,folderPath);   // initialize to start folder
    while ( toknum != -1 )                             // repeat for all files
    {
        toknum = LX_List(theDB,fName,&tokSize,0,prefix); // list next file name
        if (toknum != -1 )                             // is it a file or a folder ?
            if ( fName[tokSize-1] == ':' )             // it is a folder
                sprintf(nuPath,"%s%s",folderPath,fName); // create new folder path
                tmp = LX_SaveListContext(theDB);       // prepare for recursive listing
                myFunc(nuPath);                        // recurse!
                LX_RestoreListContext(theDB,tmp);      // restore listing context
            else                                       // it is a file...
                -- process the file somehow
    }
}
```

These routines are only a few of the routines that could be used in conjunction with the present invention. Those in the prior art will appreciate that any number of additional routines could be provided to permit manipulation of the DB and lexical analyzer. For example, the following non-exclusive list of additional routines are basic to lexical analyzer use but will not be described in detail since their implementation may be easily deduced from the basic data structures described above:

LX_Add() – Adds a new symbol to a recognizer table. The implementation of this routine is similar to LX_Lex() except when the algorithm reaches a point where the input token does not match, it then enters a second loop to append additional blocks to the recognizer table that will cause recognition of the new token.

LX_Sub() – Subtracts a symbol from a recognizer table. This consists of removing or altering table elements in order to prevent recognition of a previously entered symbol.

LX_Set() – Alters the token value for a given symbol. Basically equivalent to a call to LX_Lex() followed by assignment to the table token value at the point where the symbol was recognized.

LX_Init() – Creates a new empty recognizer DB.

LX_KillDB() – Disposes of a recognizer DB.

LX_FindToken() – Converts a token number to the corresponding token string using LX_List().

In addition to the above routines, additional routines and structures within a recognizer DB may be used to handle certain aspects of punctuation and white space that may vary between languages to be recognized. This is particularly true if a non-Roman script system is involved, such as is the case for many non-European languages. In order to distinguish between delimiter characters (i.e., punctuation etc.) and non-delimiters (i.e., alphanumeric characters), the invention may also include the routines LX_AddDelimiter() and LX_SubDelimiter(). When a recognizer DB is first created by LX_Init(), the default delimiters are set to match those used by the English language. This set can then be selectively modified by adding or subtracting the ASCII codes of interest. Whether an ASCII character is a delimiter or not is determined by whether the corresponding bit is set in a bit-array 'Dels' associated with the recognizer DB and it is this array that is altered by calls to add or subtract an ASCII code. In a similar manner, determining whether a character is white-space is crucial to determining if a given token should be recognized, particularly where a longer token with the same prefix exists (e.g., Smith and Smithsonian). For this reason, a second array 'whitespace' is associated with the recognizer DB and is used to add new whitespace characters. For example an Arabic space character has the ASCII value of the English space plus 128. This array is accessed via LX_AddDelimiter() and LX_SubDelimiter() functions.

A sample structure for a recognizer DB 500 is set forth in Figure 5. The elements of the structure 500 are as follows: onecatmax 501 (storing the number of elements in 'onecat'), catrangemax 502 (storing number of elements in 'catrange'), lexFlags 503 (storing behavior configuration options), maxToken 504 (representing the highest token number in table), nSymbols 505 (storing number of symbols in table), name 506 (name of lexical recognizer DB 500), Dels 507 (holds delimiter characters for DB), MaxAccState 508 (highest accepting state for catrange), whitespace 509 (for storing additional whitespace characters), entry 510 (storing entry points for each character), onecat 511 (a table for storing single state transitions using record type ET_onecat 200) and catrange 512 (a table storing range transitions and is record type ET_CatRange 400).

As the above description makes clear, the two-phase approach to lexical analysis provides significant advantages over standard techniques in terms of performance and flexibility when implemented in software. Additional applications are enhanced when the invention is imlemented in hardware.

Referring now to Figure 6, a sample implementation of a hardware device based on the 'OneCat' algorithm (henceforth referred to as a Single Transition Module 600 or STM 600) is shown. The STM module 600 is preferably implemented as a single chip containing a large amount of recognizer memory 605 combined with a simple bit-slice execution unit 610, such as a 2610 sequencer standard module and a control input 645. In operation the STM 600 would behave as follows:

1) The system processor on which the user program resides (not shown) would load up a recognizer DB 800 into the recognizer memory 605 using the port 615 formatted as a record of type ET_onecat 200.

2) The system processor would initialize the source of the text input stream to be scanned. The simplest external interface for text stream processing might be to tie the 'Next' signal 625 to an incrementing address generator 1020 such that each pulse on the 'Next' line 625 is output by the STM 600 and requests the system processor to send the next byte of text to the port 630. The contents of the next external memory location (previously loaded with the text to be scanned) would then be presented to the text port 630. The incrementing address generator 1020 would be reset to address zero at the same time the STM 600 is reset by the system processor.

Referring now to Figure 7, another illustration of the operation of the STM 600 is shown. As the figure illustrates, once the 'Reset' line 620 is released, the STM 600 fetches successive input bytes by clocking based on the 'Next' line 620, which causes external circuitry to present the new byte to input port 630. The execution unit 610 (as shown in Figure 6) then performs the 'OneCat' lexical analyzer algorithm described above. Other hardware implementations, via a sequencer or otherwise, are possible and would be obvious to those skilled in the art. In the simple case, where single word is to be recognized, the algorithm drives the 'Break' line 640 high at which time the state of the 'Match' line 635 determines how the external processor/circuitry 710 should interpret the contents of the table address presented by the port 615. The 'Break' signal 640 going high signifies that the recognizer (not shown) has completed an attempt to recognize a token within the text 720. In the case of a match, the contents presented by the port 615 may be used to determine the token number. The 'Break' line 640 is fed back internally within theLexical Analyzer Module or 'LAM' (see Figure 14) to cause the recognition algorithm to re-start at state zero when the next character after the one that completed the cycle is presented.

Referring now to Figure 8, a logical representation of an internal STM implementation is shown. The fields/memory described by the ET_onecat 200 structure is now represented by three registers 1110, 1120, 1130, two of 8 bits 1110, 1120 and one of at least 32 bits 1130 which are connected logically as shown. The 'Break' signal 640 going high signifies that the STM 600 has completed an attempt to recognize a token within the text stream. At this point external circuitry or software can examine the state of the 'Match' line 635 in order to decide between the following actions:

1) If the 'Match' line 635 is high, the external system can determine the token number recognized simply by examining recognizer memory 605 at the address presented via the register 1145.

2) If the 'Match' line 635 is low, then the STM 600 failed to recognize a legal token and the external system may either ignore the result, reset the STM 600 to try for a new match, or alternatively execute the range transition algorithm 500 starting from the original text point in order to determine if a token represented by a range transition exists. The choice of which option makes sense at this point is a function of the application to which the STM 600 is being applied.

The "=?" block 1150, "0?" blocks 1155, 1160, and "Add" block 1170 in Figure 11 could be implemented using standard hardware gates and circuits. Implementation of the "delim?" block 1165 would require the external CPU to load up a 256*1 memory block with 1 bits for all delimiter characters and 0 bits for all others. Once loaded, the "delim?" block

1165 would simply address this memory with the 8-bit text character 1161 and the memory output (0 or 1) would indicate whether the corresponding character was or was not a delimiter. The same approach can be used to identify white-space characters and in practice a 256*8 memory would be used thus allowing up to 8 such determinations to be made simultaneously for any given character. Handling case insensitive operation is possible via lookup in a separate 256*8 memory block.

In the preferred implementation, the circuitry associated with the 'OneCat' recognition algorithm is segregated from the circuitry/software associated with the 'CatRange' recognition algorithm. The reason for this segregation is to preserve the full power and flexibility of the distinct software algorithms while allowing the 'OneCat' algorithm to be executed in hardware at far greater speeds and with no load on the main system processor. This is exactly the balance needed to speed up the kind of CAM and text processing applications that are described in further detail below. This separation and implementation in hardware has the added advantage of permitting arrangements whereby a large number of STM modules (Fig 6 and 7) can be operated in parallel permitting the scanning of huge volumes of text while allowing the system processor to simply coordinate the results of each STM module 600. This supports the development of a massive and scaleable scanning bandwidth.

Referring now to Figure 9, a sample hardware implementation for the 'CatRange' algorithm 500 is shown. The preferred embodiment is a second analyzer module similar to the STM 600, which shall be referred to as the Range Transition Module or RTM 1200. The RTM module 1200 is preferably implemented as a single chip containing a small amount of range table memory 1210 combined with a simple bit-slice execution unit 1220, such as a 2910 sequencer standard module. In operation the RTM would behave as follows:

1) The system processor (on which the user program resides) would load up a range table into the range table memory 1210 via the port 1225, wherein the the range table is formatted as described above with reference to ET_CatRange 300.

2) Initialization and external connections, such as the control/reset line 1230, next line 1235, match line 1240 and break line 1245, are similar to those for the STM 900.

3) Once the 'Reset' line 1230 is released, the RTM 1200 fetches successive input bytes by clocking based on the 'Next' line 1235 which causes external circuitry to present the new byte to port 1250. The execution unit 1220 then performs the 'CatRange' algorithm 500. Other implementations, via a sequencer or otherwise are obviously possible.

In a complete hardware implementation of the two-phase lexical analyzer algorithm, the STM and RTM are combined into a single circuit component known as the Lexical Analyzer Module or LAM 1400. Referring now to Figure 10, a sample LAM 1400 is shown. The LAM 1400 presents a similar external interface to either the STM 600 or RTM 1200 but contains both modules internally together with additional circuitry and logic 1410 to allow both modules 600, 1200 to be run in parallel on the incoming text stream and their results to be combined. The combination logic 1410 provides the following basic functions in cases where both modules are involved in a particular application (either may be inhibited):

1) The clocking of successive characters from the text stream 1460 via the sub-module 'Next' signals 925, 1235 must be synchronized so that either module waits for the other before proceeding to process the next text character.

2) The external LAM 'Match' signals 1425 and 'Break' signals 1430 are coordinated so that if the STM module 900 fails to recognize a token but the RTM module 1200 is still processing characters, the RTM 1200 is allowed to continue until it completes. Conversly, if the RTM 1200 completes but the STM 600 is still in progress, it is allowed to continue until it completes. If the STM 600 completes and recognizes a token, further RTM 1200 processing is inhibited.

3) An additional output signal "S/R token" 1435 allows external circuitry/software to determine which of the two sub-modules 600, 1200 recognized the token and if appropriate allows the retrieval of the token value for the RTM 1200 via a dedicated location on port 1440. Alternately, this function may be achieved by driving the address latch to a dedicated value used to pass RTM 1200 results. A control line 1450 is also provided.

The final stage in implementing very high performance hardware systems based on this technology is to implement the LAM as a standard module within a large programmable gate array which can thus contain a number of LAM modules all of which can operate on the incoming text stream in parallel. On a large circuit card, multiple gate arrays of this type can be combined. In this configuration, the table memory for all LAMs can be loaded by external software and then each individual LAM is dynamically 'tied' to a particular block of this memory, much in the same manner that the ET_LexHdl structure (described above) achieves in software. Once again, combination logic similar to the combination logic 1410 utilized between STM 600 and RTM 1200 within a given LAM 1400 can be configured to allow a set of LAM modules 1400 to operate on a single text stream in parallel. This allows external software to configure the circuitry so that multiple different recognizers, each of which may relate to a particular recognition domain, can be run in parallel. This implementation permits the development and execution of applications that require separate but simultaneous scanning of text streams for a number of distinct purposes. The external software architecture necessary to support this is not difficult to imagine, as are the kinds of sophisticated applications, especially for intelligence purposes, for which this capability might find application.

Once implemented in hardware and preferably as a LAM module 1400, loaded and configured from software, the following applications (not exhaustive) can be created:

1) **Content-addressable memory (CAM).** In a CAM system, storage is addressed by name, not by a physical storage address derived by some other means. In other words, in a CAM one would reference and obtain the information on "John Smith" simply using the name, rather than by somehow looking up the name in order to obtain a physical memory reference to the corresponding data record. This significantly speeds and simplifies the software involved in the process. One application area for such a system is in ultra-high performance database search systems, such as network routing (i.e., the rapid translation of domains and IP addresses that occurs during all internet protocol routing) advanced computing architectures (i.e., non-Von Neuman systems), object oriented database systems, and similar high performance database search systems.

2) **Fast Text Search Engine.** In extremely high performance text search applications such as intelligence applications, there is a need for a massively parallel, fast search text engine that can be configured and controlled from

software. The present invention is ideally suited to this problem domain, especially those applications where a text stream is being searched for key words in order to route interesting portions of the text to other software for in-depth analysis. High performance text search applications can also be used on foreign scripts by using one or more character encoding systems, such as those developed by Unicode and specifically UTF-8, which allow multi-byte Unicode characters to be treated as one or more single byte encodings.

3) **Language Translation**. To rapidly translate one language to another, the first stage is a fast and flexible dictionary lookup process. In addition to simple one-to-one mappings, it is important that such a system flexibly and transparently handle the translation of phrases and key word sequences to the corresponding phrases. The present invention is ideally suited to this task.

Other applications. A variety of other applications based on a hardware implementation of the lexical analysis algorithm described are possible including (but not limited to); routing hierarchical text based address strings, sorting applications, searching for repetitive patterns, and similar applications.

The foregoing description of the preferred embodiment of the invention has been presented for the purposes of illustration and description. Any number of other basic features, functions, or extensions of the foregoing method and systems would be obvious to those skilled in the art in light of the above teaching. For example, other basic features that would be provided by the lexical analyzer, but that are not described in detail herein, include case insensitivity, delimiter customization, white space customization, line-end and line-start sensitive tokens, symbol flags and tagging, analyzer backup, and other features of lexical analyzers that are well-known in the prior art. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise forms disclosed. It is intended that the scope of the invention be limited not by this detailed description but rather by the claims appended hereto.

SYSTEM AND METHOD FOR MANAGING MEMORY

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

The Macintosh Operating system ("OS"), like all OS layers, provides an API where applications can allocate and de-allocate arbitrary sized blocks of memory from a heap. There are two basic types of allocation, viz: handles and pointers. A pointer is a non-relocatable block of memory in heap (referred to as *p in the C programming language, hereinafter "C"), while a handle is a non-relocatable reference to a relocatable block of memory in heap (referred to as **h in C). In general, handles are used in situations where the size of an allocation may grow, as it is possible that an attempt to grow a pointer allocation may fail due to the presence of other pointers above it. In many operating systems (including OS X on the Macintosh) the need for a handle is removed entirely as a programmer may use the memory management hardware to convert all logical addresses to and from physical addresses.

The most difficult aspect of using handle based memory, however, is that unless the handle is 'locked', the physical memory allocation for the handle can move around in memory by the memory manager at any time. Movement of the physical memory allocation is often necessary in order to create a large enough contiguous chunk for the new block size. The change in the physical memory location, however, means that one cannot 'de-reference' a handle to obtain a pointer to some structure within the handle and pass the pointer to other systems as the physical address will inevitably become invalid. Even if the handle is locked, any pointer value(s) are only valid in the current machine's memory. If the structure is passed to another machine, it will be instantiated at a different logical address in memory and all pointer references from elsewhere will be invalid. This makes it very difficult to efficiently pass references to data. What is needed, then, is a method for managing memory references such that a reference can be passed to another machine and the machine would be able to retrieve or store the necessary data even if the physical address of the data has been changed when transferred to the new machine or otherwise altered as a result of changes to the data.

APPENDIX 2

SUMMARY OF THE INVENTION

The following invention provides a method for generating a memory reference that is capable of being transferred to different machine or memory location without jeopardizing access to relevant data. Specifically, the memory management system and method of the present invention creates a new memory tuple that creates both a handle as well as a reference to an item within the handle. In the latter case, the reference is created using an offset value that defines the physical offset of the data within the memory block. If references are passed in terms of their offset value, this value will be the same in any copy of the handle regardless of the machine. In the context of a distributed computing environment, all that then remains is to establish the equivalence between handles, which can accomplished in a single transaction between two communicating machines. Thereafter, the two machines can communicate about specific handle contents simply by using offsets.

The minimum reference is therefore a tuple comprised of the handle together with the offset into the memory block, we shall call such a tuple an 'ET_ViewRef' and sample code used to create such a tuple 100 in C is provided in Figure 1. Once this tuple has been created, it becomes possible to use the ET_ViewRef structure as the basic relocatable handle reference in order to reference structures internal to the handle even when the handle may move. The price for this flat memory model is the need for a wrapper layer that transparently handles the kinds of manipulations described above during all de-referencing operations, however, even with such a wrapper, operations in this flat memory model are considerably faster that corresponding OS supplied operations on the application heap.

BRIEF DESCRIPTION OF THE FIGURES

Figure 1 illustrates sample code used to create the minimum reference 'tuple' of the present invention;

Figure 2 illustrates a drawing convention that is used to describe the interrelationship between sub-layers in one embodiment of the present invention;

Figure 3 illustrates a sample header block that may be used to practice the present invention;

Figure 4 illustrates a simple initial state for a handle containing multiple structures;

Figure 5 illustrates the type of logical relationships that may be created between structures in a handle following the addition of a new structure;

Figure 6 illustrates a sample of a handle after increasing the size of a given structure within the handle beyond its initial physical memory allocation;

Figure 7 illustrates the manner in which a handle could be adapted to enable unlimited growth to a given structure within the handle;

Figure 8 illustrates the handle after performing an undo operation;

Figure 9 illustrates a handle that has been adapted to include a time axis in the header field of the structures within the handle;

Figure 10 illustrates the manner in which the present invention can be used to store data as a hierarchical tree; and

Figure 11 illustrates the process for using the memory model to sort structures within a handle.

DETAILED DESCRIPTION

Descriptive Conventions

In order to graphically describe the architectural components and interrelations that comprise the software, this document adopts a number of formalized drawing conventions. In general, any given software aspect is built upon a number of sub-layers. Referring now to figure 2, a block diagram is provided that depicts these sub-layers as a 'stack' of blocks. The lowest block is the most fundamental (generally the underlying OS) and the higher block(s) are successive layers of abstraction built upon lower blocks. Each such block is referred to interchangeably as either a module or a package.

The first, an opaque module 200, is illustrated as a rectangular in Figure 2A. An opaque module 200 is one that cannot be customized or altered via registered plug-ins. Such a form generally provides a complete encapsulation of a given area of functionality for which customization is either inappropriate or undesirable.

The second module, illustrated as T-shaped form 210 in Figure 2B, represents a module that provides the ability to register plug-in functions that modify its behavior for particular purposes. In Figure 2A, these plug-ins 220 are shown as 'hanging' below the horizontal bar of the module 210. In such cases, the module 210 provides a complete 'logical' interface to a certain functional capability while the plug-ins 220 customize that functionality as desired. In general, the plug-ins 220 do not provide a callable API of their own. This methodology provides the benefits of customization and flexibility without the negative effects of allowing application specific knowledge to percolate any higher up the stack than necessary. Generally, most modules provide a predefined set of plug-in behaviors so that for normal operation they can be used directly without the need for plug-in registration.

In any given diagram, the visibility of lower layers as viewed from above, implies that direct calls to that layer from higher-level layers above is supported or required as part of normal operation. Modules that are hidden vertically by higher-level modules, are not intended to be called directly in the context depicted.

Figure 2C illustrates this descriptive convention. Module 230 is built upon and makes use of modules 235, 240, and 245 (as well as what may be below module 245). Module 230, 235 and 240 make use of module 245 exclusively. The functionality within module 240 is completely hidden from higher level modules via module 230, however direct access to modules 250 and 235 (but not 245) is still possible.

In Figure 2D, the Viewstructs memory system and method 250 is illustrated. The ViewStructs 250 package (which implements the memory model described herein) is layered directly upon the heap memory encapsulation 280 provided by the TBFilters 260, TrapPatches 265, and WidgetQC 270 packages. These three packages 260, 265, 270 form the heap memory abstraction, and provide sophisticated debugging and memory tracking capabilities that are discussed elsewhere. When used elsewhere, the terms ViewStructs or memory model apply only to the contents of a single handle within the heap.

To reference and manipulate variable sized structures within a single memory allocation, we require that all structures start with a standard header block. A sample header block (called an ET_Hdr) may be defined in C programming language as illustrated in Figure 3. For the purpose of discussing the memory model, we shall only consider the use of ET_Offset fields 310, 320, 330, 340. The word 'flags' 305, among other things, indicates the type of record follows the ET_Hdr. The 'version' 350 and 'date' fields 360 are associated with the ability to map old or changed structures into the latest structure definition, but these fields 350, 360 are not necessary to practice the invention and are not discussed herein.

Referring now to Figure 4, Figure 4 illustrates a simple initial state for a handle containing multiple structures. The handle contains two distinct memory structures, structure 410 and structure 420. Each structure is preceded by a header record, as previously illustrated in Figure 3, which defines its type (not shown) and its relationship to other structures in the handle. As can be seen from the diagram, the 'NextItem' field 310 is simply a daisy chain where each link simply gives the relative offset from the start of the referencing structure to the start of the next structure in the handle. Note that all references in this model are relative to the start of the referencing structure header and indicate the (possibly scaled) offset to the start of the referenced structure header. The final structure in the handle is indicated by a header record 430 with no associated additional data where 'NextItem = 0'. By following the 'NextItem' daisy chain it is possible to examine and locate every structure within the handle.

As the figure illustrates, the 'parent' field 340 is used to indicate parental relationships between different structures in the handle. Thus we can see that structure B 420 is a child of structure A 410. The terminating header record 430 (also referred to as an ET_Null record) always has a parent field that references the immediately preceding structure in the handle. Use of the parent field in the terminating header record 430 does not represent a "parent" relationship, it is simply a convenience to allow easy addition of new records to the handle. Similarly, the otherwise meaningless 'moveFrom' field 330 for the first record in the handle contains a relative reference to the final ET_Null. This provides an expedient way to locate the logical end of the handle without the need to daisy chain through the 'nextItem' fields for each structure.

Referring now to Figure 5, Figure 5 illustrates the logical relationship between the structures after adding a third structure C 510 to the handle. As shown in Figure 5, structure C 510 is a child of B 420 (grandchild of A 410). The insertion of the new structure involves the following steps:

1) If necessary, grow the handle to make room for C 510, C's header 520, and the trailing ET_Null record 430;

2) Overwrite the previous ET_Null 430 with the header and body of structure C 510.

3) Set up C's parent relationship. In the illustrated example, structure C 510 is a child of B 420, which is established by pointing the 'parent' field of C's header file 520 to the start of structure B 420.

4) Append a final ET_Null 530, with parent referenced to C's header 520.

5) Adjust the 'moveFrom' field 330 to reflect the offset of the new terminating ET_Null 530.

In addition to adding structures, the present invention must handle growth within existing structures. If a structure, such as structure B 420, needs to grow, it is often problematic since there may be another structure immediately following the one being grown (structure C 510 in the present illustration). Moving all trailing structures down to make enough room for the larger B 420 is one way to resolve this issue but this solution, in addition to being extremely inefficient for large handles, destroy the integrity of the handle contents, as the relative references within the original B structure 420 would be rendered invalid once such a shift had occurred. The handle would then have to be scanned looking for such references and altering them. The fact that structures A 410, B 420, and C 510 will generally contain relative references over and above those in the header portion make this impractical without knowledge of all structures that might be part of the handle. In a dynamic computing environment such knowledge would rarely, if ever, be available, making such a solution impractical and in many cases impossible.

For these reasons, the header for each structure further includes a moveFrom and moveTo fields. Figure 6 illustrates the handle after growing B 420 by adding the enlarged B' structure 610 to the end of the handle. As shown, the original B structure 420 remains where it is and all references to it (such as the parent reference from C 510) are unchanged. B 420 is now referred to as the "base record" whereas B' 610 is the "moved record". Whenever any reference is resolved now, the process of finding the referenced pointer address using C code is:

```
src = address of referencing structure header
dst = src + ET_Offset value for the reference
if ( dst->moveTo )
   dst = dst + dst->moveTo     -- follow the move
```

Further whenever a new reference is created, the process of finding the referenced pointer using C code is:

```
src = address of referencing structure header
dst = address of referenced structure header
if ( dst->moveFrom )
   dst = dst + dst->moveFrom;
ref value = dst - src
```

Thus, the use of the moveto and movefrom fields ensures that no references become invalid, even when structures must be moved as they grow.

Figure 7 illustrates the handle when B 420 must be further expanded into B" 710. In this case the 'moveTo' of the base record 420 directly references the most recent version of the structure, in this example B" 710. Correspondingly, the record B'' 710 now has a 'moveFrom' 720 field that references the base record 420. B's moveFrom 720 still refers back to B 420 and indeed if there were more intermediate records between B 420 and B'' (such as B' 610 in this example) the 'moveTo' and 'moveFrom' fields for all of the records 420, 610, 710 would form a doubly linked list. Once each of these records 420, 610, 710 have been linked, it is possible to re-trace through all previous versions of a structure using these links. For example, one could find all previous versions of the record starting with B" 710 by following the 'movefrom' field 720 to the base record 420 and then following the 'nextItem' link of each record until a record with a 'moveFrom' referencing the base record 420 is found. Alternatively, and perhaps more reliably, one could look for structures whose 'moveTo' field references record 420 and then work backward through the chain to find earlier versions.

This method, in which the last 'grown' structure moves to the end of the handle, has the beneficial effect that the same structure is often grown many times in sequence and in these cases we can optionally avoid creating a series of intermediate 'orphan' records. References occurring from within the bodies of structures may be treated in a similar manner to those described above and thus by extrapolation one can see that arbitrarily complex collections of cross-referencing structures can be created and maintained in this manner all within a single 'flat' memory allocation.

The price for this flat memory model is the need for a wrapper layer that transparently handles the kinds of manipulations described above during all de-referencing operations, however, even with such a wrapper, operations in this flat memory model are considerably faster that corresponding OS supplied operations on the application heap. Regardless of complexity, a collection of cross-referencing structures created using this approach is completely 'flat' and the entire 'serialization' issue is avoided when passing such collections between processors. This is a key requirement in a distributed data-flow based environment.

In addition to providing the ability to grow and move structures without impacting the references in other structures, another advantage of the 'moveTo'/'moveFrom' approach is inherent support for 'undo'. FIGURE 8 illustrates the handle after performing an 'undo' on the change from B' to B''. The steps involved for 'undo' are provided below:

```
src = base record (i.e., B)
dst = locate 'moved' record (i.e. B'') by following 'moveTo' of base record
prev = locate last record in handle whose 'moveTo' references dst
src->moveTo = prev – src;
```

The corresponding process for 'redo' (which restores the state to that depicted after B'' was first added) is depicted below:

```
src = base record (i.e., B)
```

```
dst = locate 'moved' record (i.e. B') by following 'moveTo' of base record
if ( dst->moveTo )
   nxt = dst + dst->moveTo
src->moveTo = nxt - src;
```

This process works because of the fact that 'moveTo' fields are only followed once when referencing via the base record. The ability to trivially perform undo/redo operations is very useful in situations where the structures involved represent information being edited by the user, it is also an invaluable technique for handling the effects of a time axis in the data.

One method for maintaining a time axis is by using a date field in the header of each structure. In this situation, the undo/redo mechanism can be combined with a 'date' field 910 in the header that holds the date when the item was actually changed. This process is illustrated in Figure 9 (some fields have been omitted for clarity).

This time axis can also be used to track the evolution of data over time. Rather than using the 'moveTo' fields to handle growing structures, the 'moveTo' fields could be used to reference future iterations of the data. For example, the base record could specify that it stores the high and low temperatures for a given day in Cairo. Each successive record within that chain of structures could then represent the high and low temperatures for a given date 910, 920, 930, 940. By using the 'date' fields 910, 920, 930, 940 in this fashion, the memory system and method can be used to represent and reference time-variant data, a critical requirement of any system designed to monitor, query, and visualize information over time. Moreover, this ability to handle time variance exists within the 'flat' model and thus data can be distributed throughout a system while still retaining variance information. This ability lends itself well to such things as evolving simulations, database record storage and transaction rollback, and animations.

Additionally, if each instance of a given data record represents a distinct version of the data designed for a different 'user' or process, this model can be used to represent data having multiple values depending on context. To achieve this, whatever variable is driving the context is simply used to set the 'moveTo' field of the base record, much like time was used in the example above. This allows the model to handle differing security privileges, data whose value is a function of external variables or state, multiple distinct sources for the same datum, configuration choices, user interface display options, and other multi-value situations.

A 'flags' field could also be used in the header record and can be used to provide additional flexibility and functionality within the memory model. For example, the header could include a 'flag' field that is split into two parts. The first portion could contain arbitrary logical flags that are defined on a per-record type basis. The second portion could be used to define the structure type for the data that follows the header. While the full list of all possible structure types is a matter of implementation, the following basic types are examples of types that may be used and will be discussed herein:

kNullRecord – a terminating NULL record, described above.

kStringRecord – a 'C' format variable length string record.

kSimplexRecord – a variable format/size record whose contents is described by a type-id.

kComplexRecord – a 'collection' element description record (discussed below)

kOrphanRecord – a record that has been logically deleted/orphaned and no longer has any meaning.

By examining the structure type field of a given record, the memory wrapper layer is able to determine 'what' that record is and more importantly, what other fields exist within the record itself that also participate in the memory model, and must be handled by the wrapper layer. The following definition describes a structure named 'kComplexRecord' and will be used to illustrate this method:

```
typedef struct ET_Complex      //    Collection element record
{
   ET_Hdr                            hdr;          // Standard header
   ...
   ET_Offset /* ET_SimplexPtr */     valueR;       // value reference
   ET_TypeID                         typeID;       // ID of this type
   ET_Offset /* ET_ComplexPtr */     nextElem;     // next elem. link
   ET_Offset /* ET_ComplexPtr */     prevElem;     // prev. elem. link
   ET_Offset /* ET_ComplexPtr */     childHdr;     // First child link
   ET_Offset /* ET_ComplexPtr */     childTail;    // Last child link
   ...
} ET_Complex;
```

The structure defined above may be used to create arbitrary collections of typed data and to navigate around these collections. It does so by utilizing the additional ET_Offset fields listed above to create logical relationships between the various elements within the handle.

Figure 10 illustrates the use of this structure 1010 to represent a hierarchical tree 1020. The ET_Complex structure defined above is sufficiently general, however, that virtually any collection metaphor can be represented by it including (but not limited to) arrays (multi-dimensional), stacks, rings, queues, sets, n-trees, binary trees, linked lists etc. The 'moveTo', 'moveFrom' and 'nextItem' fields of the header have been omitted for clarity. The 'valueR' field would contain a relative reference to the actual value associated with the tree node (if present), which would be contained in a record of type ET_Simplex. The type ID of this record would be specified in the 'typeID' field of the ET_Complex and, assuming the existence of an infrastructure for converting type IDs to a corresponding type and field arrangement, this could be used to examine the contents of the value (which could further contain ET_Offset fields as well).

As Figure 10 illustrates, 'A' 1025 has only one child (namely 'B' 1030), both the 'childHdr' 1035 and 'childTail' 1040 fields reference 'B' 1030, this is in contrast to the 'childHdr' 1045 and 'childTail' 1070 fields of 'B' 1030 itself which reflect the fact that 'B' 1030 has three children 1050, 1055, 1060. To navigate between children 1050, 1055, 1060, the doubly-linked 'nextItem' and 'prevItem' fields are used. Finally the 'parent' field from the standard header is used to represent the hierarchy. It is easy to see how simply by manipulating the various fields of the ET_Complex structure, arbitrary collection types can be created as can a large variety of common operations on those types. In the example of the tree above, operations might include pruning, grafting, sorting, insertion, rotations, shifts, randomization, promotion, demotion etc. Because the ET_Complex type is 'known' to the wrapper layer, it can transparently handle all the manipulations to the ET_Offset fields in order to ensure referential integrity is maintained during all such operations. This ability is critical to situations where large collections of disparate data must be accessed and distributed (while maintaining 'flatness') throughout a system.

Figure 11 illustrates the process for using the memory model to "sort" various structures. A sample structure, named ET_String 1100, could be defined in the following manner (defined below) to perform sorting on variable sized structures:

```
typedef struct ET_String                            // String Structure
{
   ET_Hdr       hdr;                                // Standard header
   ET_Offset /* ET_StringPtr */  nextString;  // ref. to next string
   ...
   char         theString[ 0 ];            // C string (size varies)
} ET_String;
```

Prior to the sort, the 'nextString' fields 1110, 1115, 1120, 1125 essentially track the 'nextItem' field in the header, indeed 'un-sort' can be trivially implemented by taking account of this fact. By accessing the strings in such a list by index (i.e., by following the 'nextString' field), users of such a 'string list' abstraction can manipulate collections of variable sized strings. When combined with the ability to arbitrarily grow the string records as described previously (using 'moveTo' and 'moveFrom'), a complete and generalized string list manipulation package is relatively easy to implement. The initial 'Start' reference 1130 in such a list must obviously come from a distinct record, normally the first record in the handle. For example, one could define a special start record format for containers describing executable code hierarchies. The specific implementation of these 'start' records are not important. What is important, however, is that each record type contain a number of ET_Offset fields that can be used as references or 'anchors' into whatever logical collection(s) is represented by the other records within the handle.

The process of deleting a structure in this memory model relates not so much to the fields of the header record itself, but rather to the fields of the full structure and the logical relationships between them. In other words, the record itself is not deleted from physical memory, rather it is logically deleted by removing from all logical chains that reference it. The specific manner in which references are altered to point "around" the deleted record will thus vary for each particular record type. Figure 12 illustrates the situation after deleting "Dog" 1125 from the string list 1100 and 'C' 1050from the tree 1020.

When being deleted, the deleted record is generally 'orphaned'. In order to more easily identify the record as deleted, a record may be set to a defined record type, such as 'kOrphanRecord'. This record type could be used during compression operations to identify those records that have been deleted. A record could also be identified as deleted by confirming that it is no longer referenced from any other structure within the handle. Given the complete knowledge that the wrapper layer has of the various fields of the structures within the handle, this condition can be checked with relative ease and forms a valuable double-check when particularly sensitive data is being deleted.

The compression process involves movement of higher structures down to fill the gap and then the subsequent adjustment of all references that span the gap to reduce the reference offset value by the size of the gap being closed during compression. Once again, the fact that the wrapper layer has complete knowledge of all the ET_Offset fields within the structures in the handle make compression a straightforward operation.

The foregoing description of the preferred embodiment of the invention has been presented for the purposes of illustration and description. For example, the term "handle" throughout this description is addressed as it is currently used in the Macintosh OS. This term should not be narrowly construed to only apply to the Macintosh OS, however, as the method and system could be used to enhance any sort of memory management system. The descriptions of the header structures should also not be limited to the embodiments described. While the defined header structures provide examples of the structures that may be used, the plurality of header structures that could in fact be implemented is nearly limitless. Indeed, it is the very flexibility afforded by the memory management system that serves as its greatest strength. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. In particular due to the simplicity of the model, hardware based implementations can be envisaged. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

A SYSTEM FOR EXCHANGING BINARY DATA

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

In most modern computer environments, such as programming languages, and applications, the programming language compiler itself performs the job of defining data structures and the types and the fields that make them up. That type information is compile-time determined. This approach has the advantage of allowing the compiler itself to detect many common programmer errors in accessing compound data structures rather than allowing such errors to occur at run-time where they are much harder to find. However, this approach is completely inadequate to the needs of a distributed and evolving system since it is impossible to ensure that the code for all nodes on the system has been compiled with a compatible set of type definitions and will therefore operate correctly. The problem is aggravated when systems from different vendors wish to exchange data and information since their type definitions are bound to be different and thus the compiler can give no help in the exchange. In recent years, technologies such as B2B suites and XML have emerged to try to facilitate the exchange of information between disparate knowledge representation systems by use of common tags, which may be used by the receiving end to identify the content of specific fields. If the receiving system does not understand the tag involved, the corresponding data may be discarded. These systems simply address the problem of converting from one 'normalized' representation to another, (i.e., how do I get it from my relational database into yours?) by use of a tagged, textual, intermediate form (e.g. XML). Such text-based markup-language approaches, while they work well for simple data objects, have major shortcomings when it comes to the interchange of complex multimedia and non-flat (i.e., having multiple cross-referenced allocations) binary data. Despite the 'buzz' associated with the latest data-interchange techniques, such systems and approaches are totally inadequate for addressing the kinds of problems faced by a system, such as an intelligence system, which attempt to monitor and capture ever-changing streams of unstructured or semi-structured inputs, from the outside world and derive knowledge, computability, and understanding from the data so gathered. The conversion of information, especially complex and multimedia information to/from a textual form such as XML becomes an unacceptable burden on complex information systems and is inadequate for describing many complex data interrelationships. This approach is the current state of the art. At a minimum, what is needed is an interchange language designed to describe and manipulate typed binary data at run-time. Ideally, this type information will be held in a 'flat' (i.e., easily transmitted) form and ideally is capable of being embedded in the data itself without impact on data integrity. The system would also ideally make use of the power of compiled strongly typed programming languages (such as C) to define arbitrarily interrelated and complex structures, while preserving the ability to use this descriptive power at run-time to interpret and create new types.

SUMMARY OF INVENTION

The present invention provides a strongly-typed, distributed, run-time system capable of describing and manipulating arbitrarily complex, non-flat, binary data derived from type descriptions in a standard (or slightly extended) programming language, including handling of type inheritance. The invention comprises four main components. First, a plurality of databases having binary type and field descriptions. The flat data-model technology (hereinafter "Claimed Database") described in Appendix 1 is the preferred model for storing such information because it is capable of providing a 'flat' (i.e., single memory allocation) representation of an inherently complex and hierarchical (i.e., including type inheritance) type and field set. Second, a run-time modifiable type compiler that is capable of generating type databases either via explicit API calls or by compilation of unmodified header files or individual type definitions in a standard programming language. This function is preferably provided by the parsing technology disclosed in Appendix 2 (hereinafter "Claimed Parser"). Third, a complete API suite for access to type information as well as full support for reading and writing types, type relationships and inheritance, and type fields, given knowledge of the unique numeric type ID and the field name/path. A sample API suite is provided below. Finally, a hashing process for converting type names to unique type IDs (which may also incorporate a number of logical flags relating to the nature of the type). A sample hashing scheme is further described below.

The system of the present invention is a pre-requisite for efficient, flexible, and adaptive distributed information systems.

BRIEF DESCRIPTION OF THE DRAWINGS

Figure 1 provides a sample implementation of the data structure ET_Field;

Figure 2 provides a sample code implementation of the data structure ET_Type;

Figure 3 is a block diagram illustrating a sample type definition tree relating ET_Type and ET_Field data structures; and

Figure 4 provides a sample embodiment of the logical flags that may be used to describe the typeID.

DETAILED DESCRIPTION OF THE INVENTION

The following description provides an overview of one embodiment of the invention. Please refer to the patent application incorporated herein for a more complete understanding of the Claimed Parser and Claimed Database.

All type information can be encoded by using just two structure variants, these are the 'ET_Field' structure, which is used to describe the fields of a given type, and the 'ET_Type' structure, which is used to described the type itself. Referring now to Figure 1, a sample implementation of the ET_Field structure 100 is provided. The fields in the ET_Field structure are defined and used as follows:

"hdr" 102 - This is a standard header record of type ET_Hdr as defined in the Claimed Database patent application.

"typeID" 104 - This field, and the union that surrounds it, contain a unique 64-bit type ID that will be utilized to rapidly identify the type of any data item. The method for computing this type ID is discussed in detail below.

"fName"106 - This field contains a relative reference to an ET_String structure specifying the name of the field.

"fDesc" 108 - This field may contain a relative reference to an ET_String structure containing any descriptive text associated with the field (for example the contents of the line comments in the type definitions above).

"fieldLink" 110 – This field contains a relative reference to the next field of the current type. Fields are thus organized into a link list that starts from the "fieldHDR" 220 field 220 of the type and passes through successive "fieldLink" 110 links 110 until there are no more fields.

"offset" 112 - This field contains the byte offset from the start of the parent type at which the field starts. This offset provides rapid access to field values at run-time.

"unitID" 114 - This field contains the unique unit ID of the field. Many fields have units (e.g., miles-per-hour) and knowledge of the units for a given field is essential when using or comparing field values.

"bounds" 116 - For fields having array bounds (e.g., and array of char[80]), this field contains the first array dimension.

"bounds2" 118 - For two dimensional arrays, this field contains the second dimension. This invention is particularly well-adapted for structures of a higher dimensionality than two, or where the connections between elements of a structure is more complex that simple array indexing.

"fScript" 120 - Arbitrary and pre-defined actions, functions, and scripts may be associated with any field of a type. These 'scripts' are held in a formatted character string referenced via a relative reference from this field.

"fAnnotation" 122 – In a manner similar to scripts, the text field referenced from this field can contain arbitrary annotations associated with the field. The use of these annotations will be discussed in later patents.

"flagIndex" 124 – It is often convenient to refer to a field via a single number rather than carrying around the field name. The field index is basically a count of the field occurrence index within the parent type and serves this purpose.

"fEchoField" 126– This field is associated with forms of reference that are not relevant to this patent and is not discussed herein.

"flagIndexTypeID" 128– In cases where a field is embedded within multiple enclosing parent types, the 'flagIndex' value stored in the field must be tagged in this manner to identify which ancestral enclosing type the index refers to.

Referring now to Figure 2, a sample embodiment of the ET_Type structure 200 is provided. The fields of the ET_Type structure 200 are defined and used as follows:

"hdr" 202 - This is a standard header record of type ET_Hdr as defined in the Claimed Database patent application.

"typeID" 204 - This field, and the union that surrounds it, contain a unique 64-bit type ID that will be utilized to rapidly identify the type of any data item. The method for computing this type ID is discussed in detail below.

"name" 206 - This is a relative reference to a string giving the name of the type.

"edit","display" 208 - These are relative references to strings identifying the "process" to be used to display/edit this type (if other than the default). For example the specialized process to display/edit a color might be a color-wheel dialog rather than a simple dialog allowing entry of the fields of a color (red,green,blue).

"description" 210- This is a relative reference to a string describing the type.

"ChildLink" 212 – For an ancestral type from which descendant types inherit, this field gives the relative reference to the next descendant type derived from the same ancestor. Type hierarchies are defined by creating trees of derived types. The header to the list of child types at any level is the "childHdr" field 218, the link between child types is the "ChildLink" field 212. Because types are organized into multiple type databases (as discussed later), there are two forms of such links: the local form and non-local form. The non-local form is mediated by type ID references, not relative references (as for the local form), and involves the fields "childIDLink" 236, "childIDHdr" 238, and "parentID" 240 (which hold the reference from the child type to its parent). The parent reference for the local form is held in the "parent" field of "hdr" 202.

"cTypedef" 216 - This field may optionally contain a relative reference to a string giving the C language type definition from which the type was created.

"childHdr" 218 - This field contains the header to the list of child types at any level .

"fieldHDR" 220 – Fields are organized into a link list that starts from the this field.

"keywords" 222 – This field contains a relative reference to a string contain key words by which the type can be looked up.

"bounds" 224, "bounds2" 226 – array dimensions as for ET_Field

"size" 228 – Total size of the type in bytes.

"color" 230– To facilitate type identification in various situations, types may be assigned inheritable colors.

"fileIndex" 232 – used to identify the source file from which the type was created.

"keyTypeID" 234 – This field is used to indicate whether this type is designated a "key" type. In a full data-flow based system, certain types are designated 'key' types and may have servers associated with them.

"nextKeyType" 246 - This field is used to link key types into a list.

"tScript" 242,"tAnnotation" 244 – These fields reference type scripts and annotations as for ET_Field 100.

"maxFieldIndex" 248 – This field contains the maximum field index value (see ET_Field 100) contained within the current type.

"numFields" 250 – This gives the total number of fields within the current type.

To illustrate the application of these structures 100, 200 to the respresentation of types and the fields within them, consider the type definitions below whereby the types "Cat" and "Dog" are both descendant from the higher level type "Mammal" (denoted by the "::" symbol similar to C++ syntax).

```
typedef struct Mammal
{
    RGBColor        hairColor;
    int32           gestation;          // in days
} Mammal;

typedef struct Dog::Mammal
{
    int32           barkVol;            // in decibels
} Dog;

typedef struct Cat::Mammal
{
    int32               purrVol;            // in decibels
} Cat;
```

Because they are mammals, both Cat and Dog inherit the fields "hairColor" and "gestationPeriod" which means the additional field(s) defined for each start immediately after the total of all inherited fields (from each successive ancestor). Referring now to Figure 3, this portion of the type definition tree when viewed as a tree of related ET_Type 200 and ET_Field 100 structures is shown. In this diagram, the vertical lines 305 linking the types 315, 320 are mediated via the "childHdr" 218 and "parent" 240 links. The horizontal line 310 linking Dog 320 and Cat 325 is mediated via "ChildLink" 242. Similarly for the field links 330, 335, 340, 345 within any given type, the fields involved are "parentID" 240, "fieldHDR" 220, and "fieldLink" 110. It is thus very obvious how one would navigate through the hierarchy in order to discover say all the fields of a given type. For example, the following sample pseudo code illustrates use of recursion to first process all inherited fields before processing those unique to the type itself.

```
void LoopOverFields (ET_Type   *aType)
{
    if ( aType->hdr.parent )
        LoopOverFields(aType->hdr.parent)
    for ( fieldPtr = aType->fieldHdr ; fieldPtr ; fieldPtr = fieldPtr->fieldLink )
        -- do something with the field
}
```

Given this simple tree structure in which type information is stored and accessed, it should be clear to any capable software engineer how to implement the algorithms setr forth in the Applications Programming Interface (API) given below. This API illustrates the nature and scope of one set of routines that provide full control over the run-time type system of this invention. This API is intended to be illustrative of the types of capabilities provided by the system of this invention and is not intended to be exhaustive. Sample code implementing the following defined API is provided in the attached Appendix A.

The routine **TM_CruiseTypeHierarchy()** recursively iterates through all the subtypes contained in a root type, call out to the provided callback for each type in the hierarchy. In the preferred embodiment, if the function 'callbackFunc' returns -1, this routine omits calling for any of that types sub-types.

The routine **TM_Code2TypeDB()** takes a type DB code (or TypeID value) and converts it to a handle to the types database to which it corresponds (if any). The type system of this invention allows for multiple related type databases (as described below) and this routine determines which database a given type is defined in.

TM_InitATypeDB() and **TM_TermATypeDB()** initialize and terminate a types database respectively. Each type DB is simply a single memory allocation utilizing a 'flat' memory model (such as the system disclosed in the Claimed Database patent application) containing primarily records of ET_Type 100 and ET_Field 200 defining a set of types and their inter-relationships.

TM_SaveATypeDB() saves a types database to a file from which it can be re-loaded for later use.

TM_AlignedCopy() copies data from a packed structure in which no alignment rules are applied to a normal output structure of the same type for which the alignment rules do apply. These non-aligned structures may occur when reading from files using the type manager. Different machine architectures and compilers pack data into structures with different rules regarding the 'padding' inserted between fields. As a result, these data structures may not align on convenient boundaries for the underlying processor. For this reason, this function is used to handle these differences when passing data between dissimilar machine architecture.

TM_FixByteOrdering() corrects the byte ordering of a given type from the byte ordering of a 'source' machine to that of a 'target' machine (normally 0 for the current machine architecture). This capability is often necessary when reading or writing data from/to files originating from another computer system. Common byte orderings supported are as follows:

• kBigEndian	-- e.g., the Macintosh PowerPC
• kLittleEndian	-- e.g., the Intel x86 architecture
• kCurrentByteOrdering	-- current machine architecture

TM_FindTypeDB() can be used to find the TypeDB handle that contains the definition of the type name specified (if any). There are multiple type DBs in the system which are accessed such that user typeDBs are consulted first, followed by system type DBs. The type DBs are accessed in the reverse order to that in which they were defined. This means that it is possible to override the definition of an existing type by defining a new one in a later types DB. Normally the containing typeDB can be deduced from the type ID alone (which contains an embedded DB index), however, in cases where only the name is known, this function deduces the corresponding DB. This routine returns the handle to containing type DB or NULL if not found. This invention allows for a number of distinct type DBs to co-exist so that types coming from different sources or relating to different functional areas may be self contained. In the preferred embodiment, these type DBs are identified by the letters of the alphabet ('A' to 'Z') yielding a maximum of 26 fixed type databases. In addition, temporary type databases (any number) can be defined and accessed from within a given process context and used to hold local or temporary types that are unique to that context. All type DBs are connected together via a linked list and types from any later database may reference or derive from types in an earlier database (the converse is not true). Certain of these type DBs may be pre-defined to have specialized meanings. A preferred list of type DBs that have specialized meanings as follows:

'A' - built-in types and platform Toolbox header files

'B' - GUI framework and environment header files

'C' - Project specific header files

'D' – Flat data-model structure old-versions DB (allows automatic adaption to type changes)

'E' - Reserved for 'proxy' types

'F' - Reserved for internal dynamic use by the environment

'I' - Project specific ontology types

TM_GetTypeID() retrieves a type's ID Number when given its name. If aTypeName is valid, the type ID is returned, otherwise 0 is returned and an error is reported.

TM_IsKnownTypeName() is almost identical but does not report an error if the specified type name cannot be found.

TM_ComputeTypeBaseID() computes the 32-bit unique type base ID for a given type name, returning it in the most significant 32-bit word of a 64-bit ET_TypeID 104. The base ID is calculated by hashing the type name and should thus be unique to all practical purposes. The full typeID is a 64-bit quantity where the base ID as calculated by this routine forms the most significant 32 bits while a variety of logical flags describing the type occupy the least significant 32-bits. In order to ensure that there is a minimal probability of two different names mapping onto the same type ID, the hash function chosen in the preferred embodiment is the 32-bit CRC used as the frame check sequence in ADCCP (ANSI X3.66, also known as FIPS PUB 71 and FED-STD-1003, the U.S. versions of CCITT's X.25 link-level protocol) but with the bit order reversed. The FIPS PUB 78 states that the 32-bit FCS reduces hash collisions by a factor of 10^-5 over the 16-bit FCS. Any other suitable hashing scheme, however, could be used. The approach allows type names to be rapidly and uniquely converted to the corresponding type ID by the system. This is an important feature if type information is to be reliably shared across a network by different machines. The key point is that by knowledge of the type name alone, a unique numeric type ID can be formed which can then be efficiently used to access information about the type, its fields, and its ancestry. The other 32 bits of a complete 64-bit type ID are utilized to contain logical flags concerning the exact nature of the type and are provided in Appendix A.

Given these type flag definitions and knowledge of the hashing algorithm involved, it is possible to define constants for the various built-in types (i.e., those directly supported by the underlying platform from which all other compound types can be defined by accumulation). A sample list of constants for the various built in types is provided in Appendix A.

Assuming that the constant definitions set forth in Appendix A are used, it is clear that the very top of the type hierarchy, the built-in types (from which all other types eventually derive), are similar to that exposed by the C language.

Referring now to Figure 4, a diagrammatic representation of a built-in type is shown (where indentation implies a descendant type). Within the kUniversalType 405, the set of direct descendants includes kVoidType 410, kScalarType 415, kStructType 420, kUnionType

425, and kFunctionType 430. kScalarType also includes descendants for handling integers 435, descendants for handling real numbers 440 and descendants for handling special case scalar values 445. Again, this illustrates only one embodiment of built-in types that may be utilized by the present system.

The following description provides a detailed summary of some of the functions that may be used in conjunction with the present invention. This list is not meant to be exhaustive nor or many of these functions required (depending upon the functionality required for a given implementation). The pseudo code associated with these functions is further illustrated in attached Appendix A. It will be obvious to those skilled in the art how these functions could be implemented in code.

Returning now to Appendix A, a function **TM_CleanFieldName()** is defined which provides a standardized way of converting field names within a type into human readable labels that can be displayed in a UI. By choosing suitable field names for types, the system can create "human readable" labels in the corresponding UI. The conversion algorithm can be implemented as follows:

1) Convert underscores to spaces, capitalizing any letter that immediately follows the underscore

2) Capitalize the first letter

3) Insert a space in front of every capitalized letter that immediately follows a lower case letter

4) Capitalize any letter that immediately follows a '.' character (field path delimiter)

5) De-capitalize the first letter of any of the following filler words (unless they start the sentence):

"an","and","of","the","or","to","is","as","a"

So for example:

"aFieldName" would become "A Field Name" as would "a_field_name"

"timeOfDay" would become "Time of Day" as would "time_of_day"

A function, such as **TM_AbbreveFieldName()**, could be used to provide a standardized way of converting field names within a type into abbreviated forms that are still (mostly) recognizable. Again, choosing suitable field names for types ensures both human readable labels in the corresponding UI as well as readable abbreviations for other purposes (such as generating database table names in an external relational database system). The conversion algorithm is as follows:

1) The first letter is copied over and capitalized.

2) For all subsequent letters:

a) If the letter is a capital, copy it over and any 'numLowerCase' lower case letters that immediately follow it.

b) If the letter follows a space or an underscore, copy it over and capitalize it

c) If the letter is '.', '[', or ']', convert it (and any immediately subsequent letters in this set) to a single '_' character, capitalize the next letter (if any). This behavior allows this function to handle field paths.

d) otherwise disgard it

So for example:

"aFieldName" would become "AFiNa" as would "a_field_name" if 'numLowerCase' was 1, it would be 'AFieNam' if it were 2

"timeOfDay" would become "TiOfDa" as would "time of day" if 'numLowerCase' was 1, it would be 'TimOfDay' if it were 2

For a field path example:

"geog.city[3].population" would become "Ge_Ci_3_Po" if 'numLowerCase' was 1

Wrapper functions, such as **TM_SetTypeEdit()**, **TM_SetTypeDisplay()**, **TM_SetTypeConverter()**, **TM_SetTypeCtypedef()**, **TM_SetTypeKeyWords()**,**TM_SetTypeDescription()** , and **TM_SetTypeColor()**, may be used set the corresponding field of the ET_Type structure 200. The corresponding 'get' functions are simply wrapper functions to get the same field.

A function, **TM_SetTypeIcon()**, may be provided that sets the color icon ID associated with the type (if specified). It is often useful for UI purposes to associate an identifiable icon with particular types (e.g., a type of occupation), this icon can be specified using TM_SetTypeIcon() or as part of the normal acquisition process. Auto-generated UI (and many other UI context) may use such icons to aid in UI clarity. Icons can also be inherited from ancestral types so that it is only necessary to specify an icon if the derived type has a sufficiently different meaning semantically in a UI context. The function **TM_GetTypeIcon()** returns the icons associated with a type (if any).

A function, such as **TM_SetTypeKeyType()**, may be used to associate a key data type (see **TM_GetTypeKeyType**) with a type manager type. By making this association, it is possible to utilize the full suite of behaviors supported for external APIs such as Database and Client-Server APIs, including creation and communication with server(s) of that type, symbolic invocation, etc. For integration with external APIs, another routine, such as **TM_KeyTypeToTypeID()**, may be used to obtain the type manager type ID corresponding to a given key data type. If there is no corresponding type ID, this routine returns zero.

Another function, **TM_GetTypeName()**, may be used to get a type's name given the type ID number. In the preferred embodiment, this function returns using the 'aTypeName' parameter, the name of the type.

A function, such as **TM_FindTypesByKeyword()**, may be used to search for all type DBs (available from the context in which it is called) to find types that contain the keywords specified in the 'aKeywordList' parameter. If matches are found, the function can allocate and return a handle to an array of type IDs in the 'theIDList' parameter and a count of the number of elements in this array as it's result. If the function result is zero, 'theIDList' is not allocated.

The function **TM_GetTypeFileName()** gets the name of the header file in which a type was defined (if any).

Given a type ID, a function, such as **TM_GetParentTypeID()**, can be used to get the ID of the parent type. If the given ID has no parent, an ID of 0 will be returned. If an error occurrs, a value of -1 will be returned.

Another function, such as **TM_IsTypeDescendant()**, may be used to determine if one type is the same as or a descendant of another. The TM_IsTypeDescendant() call could be used to check only direct lineage whereas TM_AreTypesCompatible() checks lineage and other factors in determining compatibility. If the source is a descendant of, or the same as, the target, TRUE is returned, otherwise FALSE is returned.

Another set of functions, hereinafter referred to as **TM_TypeIsPointer()**, **TM_TypeIsHandle()**, **TM_TypeIsRelRef()**, **TM_TypeIsCollectionRef()**, **TM_TypeIsPersistentRef()**, may be used to determine if a typeID represents a pointer/handle/relative etc. reference to memory or the memory contents itself (see typeID flag definitions). The routines optionally return the typeID of the base type that is referenced if the type ID does represent a pointer/handle/ref. In the preferred embodiment, when calling TM_TypeIsPtr(), a type ID that is a handle will return FALSE so the determination of whether the type is a handle, using a function such as TM_TypeIsHandle(), could be checked first where both possibilities may occur. The function **TM_TypeIsReference()** will return true if the type is any kind of reference. This function could also return the particular reference type via a paramter, such as the 'refType' parameter.

Another function, such as **TM_TypesAreCompatible()**, may be used to check if the source type is the same as, or a descendant of, the target type. In the preferred embodiment, this routine returns:

+1 If the source type is a descendant of the target type (a legal connection)

-1 If the source type is a group type (no size) and the target is descended from it (also a legal connection)

0 Otherwise (an illegal connection)

If the source type is a 'grouping' type (e.g., Scalar), i.e., it has no size then this routine will return compatible if either the source is ancestral to the target or vice-versa. This allows for data flow connections that are typed using a group to be connected to flows that are more restricted.

Additional functions, such as **TM_GetTypeSize()** and **TM_SizeOf()**, could be applied in order to return the size of the specified data type. For example, TM_GetTypeSize() could be provided with an optional data handle which may be used to determine the size of variable sized types (e.g., strings). Either the size of the type could be returned or, alternatively, a 0 could be returned for an error. TM_SizeOf() could be provided with a similar optional data pointer. It also could return the size of the type or 0 for an error.

A function, such as **TM_GetTypeBounds()**, could be programmed to return the array bounds of an array type. If the type is not an array type, this function could return a FALSE indicator instead.

The function **TM_GetArrayTypeElementOffset()** can be used to access the individual elements of an array type. Note that this is distinct from accessing the elements an array field. If a type is an array type, the parent type is the type of the element of that array. This knowledge can be used to allow assignment or access to the array elements through the type manager API.

The function **TM_InitMem()** initializes an existing block of memory for a type. The memory will be set to zero except for any fields which have values which will be initialized to the appropriate default (either via annotation or script calls – not discussed herein). The function **TM_NewPtr()** allocates and initializes a heap data pointer. If you wish to allocate a larger amount of memory than the type would imply, you may specify a non-zero value for the 'size' parameter. The value passed should be TM_GetTypeSize(...) + the extra memory required. If a type ends in a variable sized array parameter, this will be necessary in order to ensure the correct allocation. The function **TM_NewHdl()** performs a similar function for a heap data handle. The functions **TM_DisposePtr()** and **TM_DisposeHdl()** may be used to de-allocate memory allocated in this manner.

The function **TM_LocalFieldPath()** can be used to truncate a field path to that portion that lies within the specified enclosing type. Normally field paths would inherently satisfy this condition, however, there are situations where a field path implicitly follows a reference. This path truncation behavior is performed internally for most field related calls. This function should be used prior to such calls if the possibility of a non-local field path exists in order to avoid confusion. For example:

```
typedef struct t1
{
   char    x[16];
} t1;
```

```
typedef struct t2
{
    t1      y;
} t2;

then TM_LocalFieldPath(,t2,"y.x[3]",) would yield the string "y".
```

Given a type ID, and a field within that type, **TM_GetFieldTypeID()** will return the type ID of the aforementioned field or 0 in the case of an error.

The function **TM_GetBuiltInAncestor()** returns the first built-in direct (i.e., not via a reference) ancestor of the type ID given.

Two functions, hereinafter called **TM_GetIntegerValue()** and **TM_GetRealValue(),** could be used to obtain integer and real values in a standardized form. In the preferred embodiment, if the specified type is, or can be converted to, an integer value, the TM_GetIntegerValue() would return that value as the largest integer type (i.e., int64). If the specified type is, or can be converted to, a real value, TM_GetRealValue() would return that value the largest real type (i.e., long double). This is useful when code does not want to be concerned with the actual integer or real variant used by the type or field. Additional functions, such as **TM_SetIntegerValue()** and **TM_SetRealValue(),** could perform the same function in the opposite direction.

Given a type ID, and a field within that type, a function, hereinafter called **TM_GetFieldContainerTypeID(),** could be used to return the container type ID of the aforementioned field or 0 in the case of an error. Normally the container type ID of a field is identical to 'aTypeID', however, in the case where a type inherits fields from other ancestral types, the field specified may actually be contributed by one of those ancestors and in this case, the type ID returned will be some ancestor of 'aTypeID'. In the preferred embodiment, if a field path is specified via 'aFieldName' (e.g., field1.field2) then the container type ID returned would correspond to the immediate ancestor of 'field2', that is 'field1'. Often these inner structures are anonymous types that the type manager creates during the types acquisition process.

A function, hereinafter called **TM_GetFieldSize()**, returns the size, in bytes, of a field, given the field name and the field's enclosing type; 0 is returned if unsuccessful.

A function, hereinafter called **TM_IsLegalFieldPath()**, determines if a string could be a legal field path, i.e., does not contain any characters that could not be part of a field path. This check does not mean that the path actually is valid for a given type, simply that it could be. This function operates by rejecting any string that contains characters that are not either alphanumeric or in the set '[',']','_', or '.'. Spaces are allowed only between '[' and ']'.

Given an enclosing type ID, a field name, and a handle to the data, a function, hereinafter known as **TM_GetFieldValueH()**, could be used to copy the field data referenced by the handle into a new handle. In the preferred embodiment, it will return the handle storing the copy of the field data. If the field is an array of 'char', this call would append a terminating null byte. That is if a field is "char[4]" then at least a 5 byte buffer must be allocated in order to hold the result. This approach greatly simplifies C string handling since returned strings are guaranteed to be properly terminated. A function, such as **TM_GetFieldValueP()**, could serve as the pointer based equivalent. Additionally, a function such as **TM_SetFieldValue()** could be used to set a field value given a type ID, a field name and a binary object. It would also return an error code in an error.

A function, such as **TM_SetCStringFieldValue()**, could be used to set the C string field of a field within the specified type. This function could transparently handle logic for the various allowable C-string fields as follows:

1) if the field is a charHdl then:

   a) if the field already contains a value, update/grow the existing handle to hold the new value

   b) otherwise allocate a handle and assign it to the field

2) if the field is a charPtr then:

   a) if the field already contains a value:

      i) if the previous string is equal to or longer than the new one, copy new string into existing pointer

ii) otherwise dispose of previous pointer, allocate a new one and assign it

b) otherwise allocate a pointer and assign it to the field

3) if the field is a relative reference then:

a) this should be considered an error. A pointer value could be assigned to such a field prior to moving the data into a collection in which case you should use a function similar to the TM_SetFieldValue() function described above.

4) if the field is an array of char then:

a) if the new value does not fit, report array bounds error

b) otherwise copy the value into the array

A function, such as **TM_AssignToField()**, could be used to assign a simple field to a value expressed as a C string. For example, the target field could be:

a) Any form of string field or string reference;

b) A persistent or collection reference to another type; or

c) Any other direct simple or structure field type. In this case the format of the C string given should be compatible with a call to TM_StringToBinary() (described above) for the field type involved. The delimiter for TM_StringToBinary() is taken to be "," and the 'kCharArrayAsString' option (see TM_BinaryToString) is assumed.

In the preferred embodiment, the assignment logic used by this routine (when the 'kAppendStringValue' is present) would result in existing string fields having new values appended to the end of them rather than being overwritten. This is in contrast to the behavior of TM_SetCStringFieldValue() described above. For non-string fields, any values specified overwrite the previous field content with the exception of assignment to the 'aStringH' field of a collection or persistent reference with is appended if the 'kAppendStringValue' option is present. If the field being assigned is a collection reference and the 'kAppendStringValue' option is set, the contents of 'aStringPtr' could be appended to the contents of a string field. If the field being assigned is a persistent reference, the 'kAssignToRefType','kAssignToUniqueID' or 'kAssignToStringH' would be used to determine if the typeID, unique ID, or 'aStringH' field of the reference is assigned. Otherwise the assignment is to the name field. In the case of 'kAssignToRefType', the string could be assumed to be a valid type name which is first converted to a type ID. If the field is a relative reference (assumed to be to a string), the contents of 'aStringPtr' could be assigned to it as a (internally allocated) heap pointer.

Given an enclosing type ID, a field name, and a pointer to the data, a function such as **TM_SetArrFieldValue ()** could be used to copy the data referenced by the pointer into an element of an array field element into the buffer supplied. Array fields may have one, or two dimensions.

Functions, hereinafter named **TM_GetCStringFieldValueB()**, **TM_GetCStringFieldValueP()** and **TM_GetCStringFieldValueH()**, could be used to get a C string field from a type into a buffer/pointer/handle. In the case of a buffer, the buffer supplied must be large enough to contain the field contents returned. In other cases the function or program making the call must dispose of the memory returned when no longer required. In the preferred embodiment, this function will return any string field contents regardless of how is actually stored in the type structure, that is the field value may be in an array, via a pointer, or via a handle, it will be returned in the memory supplied. If the field type is not appropriate for a C string, this function could optionally return FALSE and provide an empty output buffer.

Given an enclosing type ID, a field name, and a pointer to the data, the system should also include a function, hereinafter name **TM_GetArrFieldValueP ()**, that will copy an element of an array field element's data referenced by the pointer into the buffer supplied. Array fields may have one, or two dimensions.

Simple wrapper functions, hereinafter named **TM_GetFieldBounds()**, **TM_GetFieldOffset()**, **TM_GetFieldUnits()**, and **TM_GetFieldDescription()**, could be provided in order to access the corresponding field in ET_Field 100. Corresponding 'set' functions (which are similar) could also be provided.

The function **TM_ForAllFieldsLoop()** is also provided that will iterate through all fields (and sub-fields) of a type invoking the specified procedure. This behavior is commonplace in a number of situations involving scanning the fields of a type. In the preferred embodiment, the scanning process should adhere to a common approach and as a result a function, such as this one, should be used for that purpose. A field action function takes the following form:

```
Boolean myActionFn        (                         // my field action function
          ET_TypeDBHdl           aTypeDBHdl,          // I: Type DB (NULL to default)
          ET_TypeID 104          aTypeID,             // I: The type ID
          ET_TypeID 104          aContainingTypeID,   // I: containing Type ID of field
          anonPtr                aDataPtr,            // I: The type data pointer
          anonPtr                context,             // IO:Use to pass custom context
          charPtr                fieldPath,           // I:Field path for field
          ET_TypeID 104          aFieldTypeID,        // I:Type ID for field
          int32                  dimension1,          // I:Field array bounds 1 (0 if
N/A)
          int32                  dimension2,          // I:Field array bounds 2 (0 if
N/A)
          int32                  fieldOffset,         // I:Offset of start of field
          int32                  options,             // I:Options flags
          anonPtr                internalUseOnly      // I:For internal use only
                          )                         // R:TRUE for success
```

In this example, fields are processed in the order they occur, sub-field calls (if appropriate) occur after the containing field call. If this function encounters an array field (1 or 2 dimensional), it behaves as follows:

a) The action function is first called once for the entire field with no field indexing specified in the path.

b) If the element type of the array is a structure (not a union), the action function will be invoked recursively for each element with the appropriate element index(es) reflected in the 'fieldPath' parameter, the appropriate element specific value in 'fieldOffset', and 0 for both dimension1 and dimension2.

This choice of behavior for array fields offers the simplest functional interface to the action function. Options are:

kRecursiveLoop -- If set, recurses through sub-fields, otherwise one-level only kDataPtrIsViewRef -- The 'aDataPtr' is the address of an ET_ViewRef designating a collection element

A function, hereinafter referred to as **TM_FieldNameExists()**, could be used to determine if a field with the given name is in the given type, or any of the type's ancestral types. If the field is found return it returns TRUE, otherwise it returns FALSE.

A function, hereinafter referred to as **TM_GetNumberOfFields()**, may be used to return the number of fields in a given structured type or a -1 in the case of an error. In the preferred embodiment, this number is the number of direct fields within the type, if the type contains sub-structures, the fields of these sub-structures are not counted towards the total returned by this function. One could use another function, such as TM_ForAllFieldsLoop(),to count fields regardless of level with 'kRecursiveLoop' set true and a counting function passed for 'aFieldFn' (see TM_GetTypeMaxFlagIndex).

Another function, referred to as **TM_GetFieldFlagIndex()**, can provide the 'flag index' for a given field within a type. The flag index of a field is defined to be that field's index in the series of calls that are made by the function TM_ForAllFieldsLoop() (described above) before it encounters the exact path specified. This index can be utilized as an index into some means of storing information or flags specific to that field within the type. In the preferred embodiment, these indeces include any field or type arrays that may be within the type. This function may also be used internally by a number of collection flag based APIs but may also be used by external code for similar purposes. In the event that TM_ForAllFieldsLoop() calls back for the enclosing structure field before it calls back for the fields within this enclosing structure, the index may be somewhat larger than the count of the 'elementary' fields within the type. Additionally, because field flag indeces can be easily converted to/from the corresponding field path (see TM_FlagIndexToFieldPath), they may be a useful way of referring to a specific field in a variety of circumstances that would make maintaining the field path more cumbersome. Supporting functions include the following: **TM_FieldOffsetToFlagIndex()** is a function that converts a field offset to the corresponding flag index within a type; **TM_FlagIndexToFieldPath()** is a function that converts a flag index to the corresponding field path within a type; and the function **TM_GetTypeMaxFlagIndex()** returns the maximum possible value that will be returned by TM_GetFieldFlagIndex() for a given type. This can be used for example to allocate memory for flag storage.

Another function, referred to as **TM_FieldNamesToIndeces()**, converts a comma seperated list of field names/paths to the corresponding zero terminated list of field indeces. It is often the case that the 'fieldNames' list references fields within the structure that is actually referenced from a field within the structure identified by 'aTypeID'. In this case, the index recorded in the index list will be of the referencing field, the remainer of the path is ignored. For this reason, it is possible that duplicate field indeces might be implied by the list of 'fieldNames' and as a result, this routine can also be programmed to automatically eliminate duplicates.

A function, hereinafter name **TM_GetTypeProxy()**, could be used to obtain a proxy type that can be used within collections in place of the full persistent type record and which contains a limited subset of the fields of the original type. While TM_GetTypeProxy() could take a list of field indeces, the function **TM_MakeTypeProxyFromFields()** could be used to take a comma separated field list. Otherwise, both functions would be identical. Proxy types are all descendant of the type ET_Hit and thus the first few fields are identical to those of ET_Hit. By using these fields, it is possible to determine the original persistent value to which the proxy refers. The use of proxys enables large collections and lists to be built up and fetched from servers without the need to fetch all the corresponding data, and without the memory requirements implied by use of the referenced type(s). In the preferred embodiment, proxy types are formed and used dynamically. This approach provides a key advantage of the type system of this invention and is crucial to efficient operation of complex distributed systems. Proxy types are temporary, that is, although they become known throughout the application as soon as they are defined using this function, they exist only for the duration of a given run of the application. Preferably, proxy types are actually created into type database 'E' which is reserved for that purpose (see above). Multiple proxys may also be defined for the same type having different index lists. In such a case, if a matching proxy already exists in 'E', it is used. A proxy type can also be used in place of the actual type in almost all situations, and can be rapidly resolved to obtain any additional fields of the original type. In one embodiment, proxy type names are of the form:

typeName_Proxy_n

Where the (hex) value of 'n' is a computed function of the field index list.

Another function that may be provided as part of the API, hereinafter called **TM_MakeTypeProxyFromFilter()**, can be used to make a proxy type that can be used within collections in place of the full persistent type record and which contains a limited subset of the fields of the original type. Preferably, the fields contained in the proxy are those allowed by the filter function, which examines ALL fields of the full type and returns TRUE to include the field in the proxy or FALSE to exclude the field. For more information concerning proxy types, see the discussion for the function TM_MakeTypeProxyFromFields().The only difference between this function and the function TM_MakeTypeProxyFromFields() is that TM_MakeTypeProxyFromFields() expects a commma separated field list as a parameter instead of a filter function. Another function, **TM_IsTypeProxy()**, could be used to determine if a given type is a proxy type and if so, what original persistent type it is a proxy for. Note that proxy type values start with the fields of ET_Hit and so both the unique ID and the type ID being referenced may be obtained more accurately from the value. The type ID returned by this function may be ancestral to the actual type ID contained within the proxy value itself. The type ET_Hit may be used to return data item lists from servers in a form that allows them to be uniquely identified (via the _system and _id fields) so that the full (or proxy) value can be obtained from the server later. ET_Hit is defined as follows:

```
typedef struct ET_Hit                              // list of query hits returned by a server
{
    OSType              _system;                   // system tag
    unsInt64            _id;                       // local unique item ID
    ET_TypeID 104       _type;                     // type ID
    int32               _relevance;                // relevance value 0..100
} ET_Hit;
```

The function **TM_GetNthFieldType()** gets the type of the Nth field in a structure. **TM_GetNthFieldName()** obtains the corresponding field name and **TM_GetNthFieldOffset()** the corresponding field offset.

Another function that may be included within the API toolset is a function called **TM_GetTypeChildren()**. This function produces a list of type IDs of the children of the given type. This function allocates a zero terminated array of ET_TypeID 104's and returns the address of the array in 'aChildIDList'; the type ID's are written into this array. If 'aChildIDList' is specified as NULL then this array is not allocated and the function merely counts the number of children; otherwise 'aChildIDList' must be the address of a pointer that will point at the typeID array on exit. A negative number is returned in the case of an error. In the preferred embodiment, various specialized options for omitting certain classes of child types are supported.

A function, hereinafter referred to as **TM_GetTypeAncestors()**, may also be provided that produces a list of type IDs of ancestors of the given type. This function allocates a zero terminated array of ET_TypeID 104 and returns the address of the array in 'ancestralIDs'; the type ID's are written into this array. If 'ancestralIDs' is specified as NULL then this array is not allocated and the function merely counts the number of ancestors; otherwise 'ancestralIDs' must be the address of a pointer that will point at the typeID array on exit. The last item in the list is a 0, the penultimate item is the primal ancestor of the given type, and the first item in the list is the immediate predecessor, or parent, of the given type. The function **TM_GetTypeAncestorPath()** produces a ':' seperated type path from a given ancestor to a descendant type. The path returned is exclusive of the type name but inclusive of the descendant, empty if the two are the same or 'ancestorID' is not an ancestor or 'aTypeID'. The function **TM_GetInheritanceChain()** is very similiar to TM_GetTypeAncestors() with the following exceptions:

(1) the array of ancestor type ids returned is in reverse order with the primal ancestor being in element 0

(2) the base type from which the list of ancestors is determined is included in the array and is the next to last element (array is 0 terminated)

(3) the count of the number of ancestors includes the base type

In the preferred embodiment, this function allocates a zero terminated array of ET_TypeID 104's and returns the address of the array in 'inheritanceChainIDs'; the type ID's are written into this array. If 'inheritanceChainIDs' is specified as NULL then this array is not allocated and the function merely counts the number of types in the inheritance chain; otherwise 'inheritanceChainIDs' must be the address of a pointer that will point at the typeID array on exit. The last item in the list is 0, element 0 is the primal ancestor of the base type, and the next to last item in the list is the base type.

The API could also include a function, hereinafter called **TM_GetTypeDescendants()**, that is able to create a tree collection whose root node is the type specified and whose branch and leaf nodes are the descendant types of the root. Each node in the tree is named by the type name and none of the nodes contain any data. Collections of derived types can serve as useful frameworks onto which various instances of that type can be 'hung' or alternatively as a navigation and/or browsing framework. The resultant collection can be walked using the collections API (discussed in a later patent). The function **TM_GetTypeSiblings()** produces a list of type IDs of sibling types of the given type. This function allocates a zero terminated array of ET_TypeID 104's and returns the address of the array in 'aListOSibs', the type ID's are written into this array. If 'aListOSibs' is specified as NULL then this array is not allocated and the function merely counts the number of siblings; otherwise 'ancestralIDs' must be the address of a pointer that will point at the typeID array on exit. The type whose siblings we wish to find is NOT included in the returned list. The function **TM_GetNthChildTypeID()** gets the n'th child Type ID for the passed in parent. The function returns 0 if successful, otherwise it returns an error code.

The function **TM_BinaryToString()** converts the contents of a typed binary value into a C string containing one field per delimited section. During conversion, each field in turn is converted to the equivalent ASCII string and appended to the entire string with the specified delimiter sequence. If no delimiter is specified, a new-line character is used. The handle, 'aStringHdl', need not be empty on entry to this routine in which case the output of this routine is appended to whatever is already in the handle. If the type contains a variable sized array as its last field (i.e., stuff[]), it is important that 'aDataPtr' be a true heap allocated pointer since the pointer size itself will be used to determine the actual dimensions of the array. In the preferred embodiment, the following specialized options are also available:

kUnsignedAsHex	-- display unsigned numbers as hex
kCharArrayAsString	-- display char arrays as C strings
kShowFieldNames	-- prefix all values by fieldName:
kOneLevelDeepOnly	-- Do Not go down to evaluate sub-structures:

An additional function, hereinafter referred to as **TM_StringToBinary()**, may also be provided in order to convert the contents of a C string of the format created by TM_BinaryToString() into the equivalent binary value in memory.

The API may also support calls to a function, hereinafter referred to as **TM_LowestCommonAncestor()**, which obtains the lowest common ancestor type ID for the two type IDs specified. If either type ID is zero, the other type ID is returned. In the event that one type is ancestral to the other, it is most efficient to pass it as the 'typeID2' parameter.

Finally, a function, referred to as **TM_DefineNewType()**, is disclosed that may be used to define a new type to be added to the specified types database by parsing the C type definition supplied in the string parameter. In the preferred embodiment, the C syntax typedef string is preserved in its entirety and attached to the type definition created so that it may be subsequently recalled. If no parent type ID is supplied, the newly created type is descended directly from the appropriate group type (e.g., structure, integer, real, union etc.) the typedef supplied must specify the entire structure of the type (i.e., all fields). If a parent type ID is supplied, the new type is created as a descendant of that type and the typedef supplied specifies only those fields that are additional to the parental type, NOT the entire type. This function is the key to how new types can be defined and incorporated into the type system at run time and for that reason is a critical algorithm to the present invention. The implementation is based on the parser technology described in Claimed Parser patent application and the lexical analyzer technology (the"Claimed Lexical Analyzer") as provided in Appendix 3. As set forth above, those pending applications are fully incorporated herein. The reader is referred to those patents (as well as the Claimed Database patent application) for additional details. The BNF specification to create the necessary types parser (which interprets an extended form of the C language declaration syntax) is provided in Appendix A. The corresponding lexical analyzer specification is also provided in Appendix A.

As can be seen from the specifications in Appendix A, the types acquisition parser is designed to be able to interpret any construct expressible in the C programming language but has been extended to support additional features. The language symbols associated with these extensions to to C are as follows:

script	-- used to associate a script with a type or field
annotation	-- used to associate an annotation with a type or field

@	-- relative reference designator (like '*' for a pointer)
@@	-- collection reference designator
#	-- persistent reference designator
<on>	-- script and annotation block start delimiter
<no>	-- script and annotation block end delimiter
><	-- echo field specification operator

In order to complete the types acquisition process, a 'resolver' function and at least one plug-in are provided. A pseudo code embodiment of one possible resolver is set forth in Appendix A. Since most of the necessary C language operations are already provided by the built-in parser plug-in zero, the only extention of this solution necessary for this application is the plug-in functionality unique to the type parsing problem itself. This will be referred to as plug-in one and the pseudo code for such a plug in is also provided in Appendix A.

The foregoing description of the preferred embodiments of the invention has been presented for the purposes of illustration and description. For example, although described with respect to the C programming language, any programming language could be used to implement this invention. Additionally, the claimed system and method should not be limited to the particular API disclosed. The descriptions of the header structures should also not be limited to the embodiments described. While the sample pseudo code provides examples of the code that may be used, the plurality of implementations that could in fact be developed is nearly limitless. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

SYSTEM AND METHOD FOR AUTOMATIC GENERATION OF SOFTWARE PROGRAMS

Inventor: John Fairweather

BACKGROUND OF THE INVENTION

In any complex information system that accepts unstructured or semi-structured input (such as an intelligence system) for the external work, it is obvious that change is the norm, not the exception. Media and data streams are often modified and otherwise constantly change making it difficult to monitor them. Moreover, in any system involving multiple users with divergent requirements, even the data models and requirements of the system itself will be subject to continuous and pervasive change. By some estimates, more than 90% of the cost and time spent on software is devoted to maintenance and upgrade of the installed system to handle the inevitability of change. Even our most advanced techniques for software design and implementation fail miserably as the system is scaled or is otherwise changed. The reasons for this failure arise, at least in part, from the very nature of accepted software development practice/process.

Referring now to Figure 1, the root of the problem with the current software development process, which we shall call the "Software Bermuda Triangle" effect, is shown. Conventional programming wisdom holds that during the design phase of an information processing application, programming teams should be split into three basic groups. The first group is labeled DBA (for Database Administrator) 105. These individuals 105 are experts in database design, optimization, and administration. This group 105 is tasked with defining the database tables, indexes, structures, and querying interfaces based initially on requirements, and later, on requests primarily from the applications group. These individuals 105 are highly trained in database techniques and tend naturally to pull the design in this direction, as illustrated by the small outward pointing arrow. The second group is the Graphical User Interface (GUI) group 110. The GUI group 110 is tasked with implementing a user interface to the system that operates according the customer's expectations and wishes and yet complies exactly with the structure of the underlying data (provided by the DBA group 105) and the application(s) behavior (as provided by the Apps group 115). The GUI group 110 will have a natural tendency to pull the design in the direction of richer and more elaborate user interfaces. Finally the applications group 115 is tasked with implementing the actual functionality required of the system by interfacing with both the DBA and the GUI and related Applications Programming Interfaces (APIs). This group 115, like the others 105,110 tends to pull things in the direction or more elaborate system specific logic. Each of these groups tends to have no more than a passing understanding of the issues and needs of the other groups. Thus during the initial design phase, assuming a strong project and software management process rigidly enforces design procedures, a relatively stable triangle is formed where the strong connections 120, 125, 130 enforced between each group by management are able to overcome the outward pull of each member of the triangle. Assuming a stable and unchanging set of requirements, such a process stands a good chance of delivering a system to the customer on time.

The problem, however, is that while correct operation has been achieved by each of the three groups 110, 105, 115 in the original development team, significant amounts of undocumented application, GUI, and Database specific knowledge has likely been embedded into all three of the major software components. In other words, this process often produces a volatile system comprised of these subtle and largely undocumented relationships just waiting to be triggered. After delivery (the bulk of the software life cycle), in the face of the inevitable changes forced on the system by the passage of time, the modified system begins to break down to yield a new "triangle" 150. Unfortunately, in many cases, the original team that built the system has disbanded and knowledge of the hidden dependencies is gone. Furthermore, system management is now in a monitoring mode only meaning that instead of having a rigid framework, each component of the system is now more likely to "drift". This drift is graphically represented by the dotted lines 155, 160, 165. During maintenance and upgrade phases, each change hits primarily one or two of the three groups. Time pressures, and the new development environment, mean that the individual tasked with the change (probably not an original team member) tends to be unaware of the constraints and naturally pulls outward in his particular direction. The binding forces have now become much weaker and more elastic while the forces pulling outwards have become much stronger. A steady supply of such changes impacting this system could well eventually break it apart. In such a scenario, the system will grind to a halt or become unworkable or un-modifiable. The customer must either continue to pay progressively more and more outrageous maintenance costs (swamping the original development costs), or must start again from scratch with a new system and repeat the cycle. The latter approach is often much cheaper than the former. This effect is central to why software systems are so expensive. Since change of all kinds is particularly pervasive in an intelligence system, any architecture for such systems would preferably address a way to eliminate this "Bermuda Triangle" effect.

Since application specific logic and it's implementation cannot be eliminated, what is needed is a system and environment in which the 'data' within the system can be defined and manipulated in terms of a world model or Ontology, and for which the DBA and GUI portions of the programming tasks can be specified and automatically generated from this Ontology thereby eliminating the triangle effect (and the need for the associated programming disciplines). Such an approach would make the resultant system robust and adaptive to change.

SUMMARY OF INVENTION

The present invention provides a system capable of overcoming this effect and provides a system that is both robust and adaptive to change. The preferred base language upon which this system is built is the C programming language although other languages may be used. In the standard embodiment using the C programing language, the present invention is composed of the following components:

a) Extensions to the language that describe and abstract the logic associated with interacting with external 'persistent' storage (i.e., non-memory based). Standard programming languages do not provide syntax or operators for manipulating persistent storage and a formalization of this capability is desirable. This invention provides these extensions and the "extended" language is henceforth referred to as C*. C*, in addition to being a standard programming language, is also an ontology definition language (ODL).

b) Extensions to the C* language to handle type inheritance. In an ontology based system, the world with which the system interacts is broken down based on the kinds of things that make up that world, and by knowledge of the kind of thing involved, it becomes possible to perform meaningful calculations on that object without knowledge of the particulars of the descendant type. Type inheritance in this context therefore more accurately means ancestral field inheritance (as will be described later).

c) Extensions to the C* language to allow specification of the GUI content and layout.

d) Extensions to the C* language to allow specification and inheritance of scriptable actions on a per-field and per-type basis. Similar extensions to allow arbitrary annotations associated with types and fields are also provided.

e) A means whereby the data described in the C* language can be translated automatically into generating the corresponding tables and fields in external databases and the queries and actions necessary to access those databases and read/write to them. This aspect of the invention enables dynamic creation of databases as data is encountered

f) A high level ontology designed to facilitate operation of the particular application being developed. In the examples below and in the preferred embodiment, the application being developed will address the problem of 'intelligence' i.e., the understanding of 'events' happening in the world in terms of the entities involved, their motives, and the disparate information sources from which reports are obtained.

g) A means to tie types and their access into a suite of federated type or container/engine specific servers responsible for the actual persistence of the data.

A necessary prerequisite for tackling the triangle problem is the existence of a run-time accessible (and modifiable) types system capable of describing arbitrarily complex binary structures and the references between them. In the preferred embodiment, the invention uses the system has been previously described in Appendix 1 (hereinafter, the "Types Patent"). Another prerequisite is a system for instantiating, accessing and sharing aggregates of such typed data within a standardized flat memory model and for associating inheritable executable and/or interpreted script actions with any and all types and fields within such data. In the preferred embodiment, the present invention uses the system and method that is described in Appendix 2 (hereinafter, the "Memory Patent"). The material presented in these two patents are expressly incorporated herein. Additional improvements and extensions to this system will also be described below and many more will be obvious to those skilled in the art.

BRIEF DESCRIPTION OF THE FIGURES

Figure 1 shows the root of the problem with the current software development process, which we shall call the “Software Bermuda Triangle” effect.

Figure 2 shows a sample query-building user interface (UI).

Figure 3 shows a sample user interface providing access to the fields within the type "country."

Figure 4 shows a sample user interface providing access to a free format text field within the type "country."

Figure 5 shows a sample user interface providing access to a fixed sized text field within the type "country."

Figure 6A shows an example of how a short text field or numeric field (such as those handled by the RDBMS container described above) might be displayed in a control group.

Figure 6B shows one method for displaying a date in a control group.

Figure 6C shows an example of an Islamic Hijjrah calendar being displayed.

Figure 7A shows the illustrated control group of how one might display and interact with a persistent reference field (‘#’).

Figure 7B shows an example of one way that a collection reference field (‘@@’) might be displayed in an auto-generated user interface.

Figure 8 shows one possible method for displaying variable sized text fields (referenced via the char @ construct).

Figure 9 shows the manner in which an image reference (Picture @picture) field could be displayed in an auto-generated user interface.

Figure 10 shows a sample screen shot of one possible display of the Country record in the same UI layout theme described above (most data omitted).

Figure 11 shows a sample embodiment of the geography page within Country.

Figure 12 shows a sample embodiment of the second sub-page of the geography page within country.

Figure 13 shows an example of one part of a high-level ontology targeted at intelligence is shown.

DETAILED DESCRIPTION OF THE INVENTION

As described above, a necessary prerequisite for tackling the triangle problem is the existence of a run-time accessible (and modifiable) types system capable of describing arbitrarily complex binary structures and the references between them. In the preferred embodiment, the invention uses the system described in the Types Patent. Another prerequisite is a system for instantiating, accessing and sharing aggregates of such typed data within a standardized flat memory model and for associating inheritable executable and/or interpreted script actions with any and all types and fields within such data. In the preferred embodiment, the present invention uses the system and method that is described in the Memory Patent. The material presented in these two patents are expressly incorporated herein and the functions and features of these two systems will be assumed for the purposes of this invention..

As an initial matter, it is important to understand some of the langauge extentions that are needed in order to create an Ontology Description Language (ODL). In the preferred embodiment, the following operators/symbols are added to the basic C language (although other symbols and syntax are obviously possible without changing the basic nature of the approach) in order to provide basic support for the items described herein:

script	-- used to associate a script with a type or field
annotation	-- used to associate an annotation with a type or field
@	-- relative reference designator (like '*' for a pointer)
@@	-- collection reference designator
#	-- persistent reference designator
<on>	-- script and annotation block start delimiter
<no>	-- script and annotation block end delimiter
><	-- echo field specification operator
:	-- type inheritance

Additionally, the syntax for a C type definition has been extended to include specification of the "key data-type" associated with a given ontological type as follows:

```
typedef struct X 'XXXX' { ... };
```

Where the character constant 'XXXX' specifies the associated key data-type. The persistent reference designator '#' implies a singular reference to an item of a named type held in external storage. Such an item can be referenced either by name or by unique system-wide ID and given this information, the underlying substrate is responsible for obtaining the actual data referenced, adding it to the collection, and making the connection between the referencing field and the newly inserted data by means of a relative reference embedded within the persistent reference structure. Preferably, the binary representation of a persistent reference field is acomplished using a structure of type 'ET_PersistentRef' as defined below:

```
typedef struct ET_UniqueID
{
   OSType              system;                 // system id is 32 bits
   unsInt64            id;                     // local id is 64 bits
} ET_UniqueID;

typedef struct ET_PersistentRef
{
   ET_CollectionHdl    members;                // member collection
   charHdl             stringH;                // String containing mined
text
   ET_TypeID           aTypeID;                // type ID
   ET_Offset           elementRef;             // rel. ref. to data (NULL
if !fetched)
   ET_Offset           memberRef;              // rel. ref. to member
coll. (or NULL)
   anonPtr             memoryRef;              // pointer to type data
(NULL if N/A)
   ET_UniqueID         id;                     // unique ID
   char                name[kPersRefNameSize]; // name of reference
} ET_PersistentRef, *ET_PersistentRefPtr;
```

The type ET_UniqueID consists of a two part 96-bit reference where the 64-bit 'id' field refers to the unique ID within the local 'system' which would normally be a single logical installation such as for a particular corporation or organization. Multiple systems can exchange data and reference between each other by use of the 32-bit 'system' field of the unique ID. The 'members' field of an ET_PersistentRef is used by the system to instantiate a collection of the possible items to which the reference is being made and this is utilized in the user interface to allow the user to pick from a list of possibilities. Thus for example if the persistent reference were "Country #nationality" then the member collection if retrieved would be filled with the names of all possible countries from which the user could pick one which would then result in filling in the additional fields required to finalize the persistent reference.

In normal operation, either the name or ID and type is known initially and this is sufficient to determine the actual item in persistent storage that is being referenced which can then be fetched, instantiated in the collection and then referenced using the 'elementRef' field. The contents of the 'stringH' field are used during data mining to contain additional informating relating to resolving the reference. The 'aTypeID' field initially takes on the same value as the field type ID from which the reference is being made, however, once the matching item has been found, a more specific type ID may be assigned to this field. For example if the referencing field were of the form "Entity #owner" (a reference to an owning entity which might be a person, organization, country etc.) then after resolution, the 'aTypeID' field would be altered to reflect the actual sub-type of entity, in this case the actual owning entity. The 'memoryRef' field might contain a heap data reference to the actual value of the referenced object in cases where the referenced value is not to become part of the containing collection for some reason. Normally however, this field is not needed.

As an example of how the process of generating and then resolving a persistent reference operates, imagine the system has just received a news story referring to an individual who's name is "X", additionally from context saved during the mining process, the system may know such things as where "X" lives and this information could be stored in the 'stringH' field. At the time the reference to "X" is instantiated into persistent storage, a search is made for a person named "X" and, should multiple people called "X" be found in the database, the information in 'stringH' would be used in a type dependant manner to prune the list down to the actual "X" that is being referenced. At this point the system-wide ID for the specific individual "X" is known (as is whatever else the system knows about X) and thus the 'id' field of the reference can be filled out and the current data for "X" returned and referenced via "elementRef". If no existing match for "X" is found, a new "Person" record for "X" is created and the unique ID assigned to that record is returned. Thus it can be seen that, unlike a memory reference in a conventional programming language, a persistent reference may go through type specific resolution processes before it can be fully resolved. This need for a 'resolution' phase is characteristic of all references to persistent storage.

Like a persistent reference, the collection reference '@@' involves a number of steps during instantiation and retrieval. In the preferred embodiment, a collection reference is physically (and to the C* user transparently) mediated via the 'ET_CollectionRef' type as set forth below:

```
typedef struct ET_CollectionRef
{
   ET_CollectionHdl    collection;          // member collection
   charHdl             stringH;             // String containing mined text
   ET_TypeID           aTypeID;             // collection type ID (if any)
   ET_Offset           elementRef;          // relative reference to collection root
   ET_StringList       cList;               // collection member list (used for UI)
} ET_CollectionRef, *ET_CollectionRefPtr;
```

The first four fields of this structure have identical types and purposes to those of the ET_PersistentRef structure, the only difference being that the 'collection' field in this structure references the complete set of actual items that form part of the collection. The 'cList' field is used internally for user interface purposes. The means whereby the collections associated with a particular reference can be distinguished from those relating to other similar references is related to the meaning and use of the 'echo field' operator '><'. The following extracts from an actual ontology based on this system serve to reveal the relationship between the '><' operator and persistent storage references:

```
typedef struct Datum     'DTUM'                   // Ancestral type of all
pers. storage
{
   NumericID             hostID;                  // unique Host system ID
(0=local)
   unsInt64              id;                      // unique ID
   char                  name[256];               // full name of this
Datum
   char                  datumType[32];           // the type of the datum
   NumericID             securityLevel;           // security level
   char                  updatedBy[30];            // person
updating/creating this Datum
   Date                  dateEntered;              // date first entered
   Date                  dateUpdated;              // date of last update
   Feed                  #source;                  // information source
for this Datum
   Language              #language;                // language for this
Datum record
   struct
   {
      NoteRegarding      @@notes >< regarding;     // Notes regarding this
Datum
      NoteRelating       @@relatedTo >< related;   // Items X-referencing
this Datum
      NoteRelating       @@relatedFrom >< regarding;// Items X-referencing
this Datum
      GroupRelation      @@relatedToGroup >< related;// Groups X-referencing
this Datum
      GroupRelation      @@relatedFromGroup >< regarding;// Groups X-
referencing Datum
      Delta              @@history >< regarding;   // Time history of
changes to Datum
      Category           @@membership;             // Groupings Datum is a
member of
      char               @sourceNotes;             // notes information
source(s)
      unsInt64           sourceIDref;              // ID reference in
original source
   } notes;
   Symbology             #symbology;               // symbology used
   Place                 #place;                   // 'where' for the datum
(if known)
} Datum , *DatumPtr;

typedef struct NoteRelating:Observation   'CXRF'   // Relationship between
two datums
{
   Datum      #regarding >< notes.relatedFrom;     // 'source' item
   char                  itemType[64];             // Datum type for
regarding item
   Datum      #related >< notes.relatedTo;         // 'target' item
   char                 relatedType[64];          // Datum type for
related item
   RelationType         #relationType;            // The type of the
relationship
   Percent              relevance;                // strength of
relationship (1..100)
   char                 author[128];              // Author of the StickIt
Relating note
   char                 title[256];               // Full Title of StickIt
Relating note
   char                 @text;                    // descriptive text and
notes
} NoteRelating;
```

In the preferred embodiment, 'Datum' is the root type of all persistent types. That is, every other type in the ontology is directly or indirectly derived from Datum and thus inherits all of the fields of Datum. The type 'NoteRelating' (a child type of Observation) is the ancestral type of all notes (imagine them as stick-it notes) that pertain to any other datum. Thus an author using the system may at any time create a note with his observations and opinions regarding any other item/datum held in the system. The act of creating such a note causes the relationships between the note and the datum to which it pertains to be written to and persisted in external storage. As can be seen, every datum in the system contains within its 'notes' field a sub-field called 'relatedFrom' declared as "NoteRelating @@relatedFrom >< regarding". This is interpreted by the system as stating that for any datum, there is a collection of items of type 'NoteRelating' (or a derived type) for which the 'regarding' field of each 'NoteRelating' item is a persistent reference to the particular Datum involved. Within each such 'NoteRelating' item there is a field 'relating' which contains a reference to some other datum that is the original item that is related to the Datum in question. Thus the 'NoteRelating' type is serving in this context as a bi-directional link relating any two items in the system as well as associating with that relationship a 'direction', a relevance or strength, and additional information (held in the @text field which can be used to give an arbitrary textual description of the exact details of the relationship). Put another way, in order to discover all elements in the 'relatedFrom' collection for a given datum, all that is necessary is to query storage/database for all 'NoteRelating' items having a 'regarding' field which contains a reference to the Datum involved. All of this information is directly contained within the type definition of the item itself and thus no external knowledge is required to make connections between disparate data items. The syntax of the C* declaration for the field, therefore, provides details about exactly how to construct and execute a query to the storage container(s)/database that will retrieve the items required. Understanding the expressive power of this syntax is key to understanding how it is possible via this methodology to eliminate the need for a conventional database administrator and/or database group to be involved in the construction and maintenance of any system built on this methodology.

As can be seen above, the 'regarding' field of the 'NoteRelating' type has the reverse 'echo' field, i.e., "Datum #regarding >< notes.relatedFrom;". This indicates that the reference is to any Datum or derived type (i.e., anything in the ontology) and that the "notes.relatedFrom" collection for the referenced datum should be expected to contain a reference to the NoteRelating record itself. Again, it is clear how, without any need for conventional database considerations, it is possible for the system itself to perform all necessary actions to add, reference, and query any given 'NoteRelating' record and the items it references. For example, the 'notes.relatedTo' field of any datum can reference a collection of items that the current datum has been determined to be related to. This is the other end of the 'regarding' link discussed above. As the type definitions above illustrate, each datum in the present invention can be richly cross referenced from a number of different types (or derivatives). More of these relationship types are discussed further herein.

For the purposes of illustrating how this syntax might translate into a concrete system for handling references and queries, it will assumed in the discussion below that the actual physical storage of the data occurs in a conventional relational database. It is important to understand, however, that nothing in this approach is predicated on or implies, the need for a relational database. Indeed, relational databases are poorly suited to the needs of the kinds of system to which the technology discussed is targeted and are not utilized in the preferred embodiment. All translation of the syntax discussed herein occurs via registered script functions (as discussed further in the Collections Patent) and thus there is no need to hard code this system to any particular data storage model so that the system can be customized to any data container or federation of such containers. For clarity of description, however, the concepts of relational database management systems (RDBMS) and how they work will be used herein for illustration purposes.

Before going into the details of the behavior of RDBMS plug-in functions, it is worth examining how the initial connection is made between these RDBMS algorithms and functions and this invention. As mentioned previously, this connection is preferably established by registering a number of logical functions at the data-model level and also at the level of each specific member of the federated data container set. The following provides a sample set of function prototypes that could apply for the various registration processes:

```
Boolean DB_SpecifyCallBack (                        // Specify a persistent storage
callback
                short            aFuncSelector,      // I:Selector for the logical
function
                ProcPtr         aCallBackFn         // I:Address of the callback
function
                                )                    // R:TRUE for success, FALSE
otherwise

#define kFnFillCollection         1              // ET_FillCollectionFn -
                                 // Fn. to fill collection with data for a given a hit
list
#define kFnFetchRecords           2              // ET_FetchRecordsFn -
                                 // Fn. to query storage and fetch matching records to
colln.
#define kFnGetNextUniqueID        3              // ET_GetUniqueIdFn -
                                 // Fn. to get next unique ID from local persistent
storage
#define kFnStoreParsedDatums      4              // ET_StoreParsedDatumsFn -
                                 // Fn. to store all extracted data in a collection
#define kFnWriteCollection        5              // ET_WriteCollectionFn -
                                 // Fn. to store all extracted data in a collection
#define kFnDoesIdExist            6              // ET_DoesIdExistFn -
                                 // Fn. to determine if a given ID exists in
persistent storage
#define kFnRegisterID             7              // ET_RegisterIDFn -
                                 // Fn. to register an ID to persistent storage
#define kFnRemoveID               8              // ET_RemoveIDFn -
                                 // Fn. to remove a given ID from the ID/Type
registery
#define kFnFetchRecordToColl      9              // ET_FetchRecordToCollFn -
                                 // Fn. Fetch a given persistent storage item into a
colln.
#define kFnFetchField             10             // ET_FetchFieldFn -
                                 // Fn. Fetch a single field from a single persistent
record
```

```
#define kFnApplyChanges            11              // ET_ApplyChangesFn -
                               // Fn. to apply changes
#define kFnCancelChanges           12              // ET_CancelChangesFn -
                               // Fn. to cancel changes
#define kFnCountTypeItems          13              // ET_CountItemsFn -
                               // Fn. to count items for a type (and descendant
types)
#define kFnFetchToElements         14              // ET_FetchToElementsFn -
                               // Fn. to fetch values into a specified set of
elements/nodes
#define kFnRcrsvHitListQuery       15              // ET_RcrsvHitListQueryFn -
                               // Fn. create a hit list from a type and it's
descendants
#define kFnGetNextValidID          16              // ET_GetNextValidIDFn -
                               // Fn. to find next valid ID of a type after a given
ID


Boolean DB_DefineContainer (                          // Defines a federated
container
             .charPtr              name              // I: Name of container
                            );                       // R: Error code (0 = no
error)

Boolean DB_DefinePluginFunction(                      // Defines container plugin
fn.
             charPtr               name,             // I: Name of container
             int32                 functionType,    // I: Which function type
             ProcPtr             functionAddress // I: The address of the function
                            );                       // R: Void

#define kCreateTypeStorageFunc              29    // Create storage for a container
#define kInsertElementsFunc                   30    // insert container data
#define kUpdateRecordsFromElementsFunc        31    // update container from data
#define kDeleteElementsFunc                   32    // delete elements from
container
#define kFetchRecordsToElementsFunc           33    // fetch container data
#define kInsertCollectionRecordFunc           34    // insert container data to
elements
#define kUpdateCollectionRecordFunc           35    // update collection from
container
#define kDeleteCollectionRecordFunc .         36    // delete collection record
#define kFetchRecordsToCollectionFunc         37    // fetch container record to
colln.
```

```
#define kCheckFieldType                          38     // determine if field is
container's
```

In this embodiment, whenever the environment wishes to perform any of the logical actions indicated by the comments above, it invokes the function(s) that have been registered using the function DB_SpecifyCallBack() to handle the logic required. This is the first and most basic step in disassociating the details of a particular implementation from the necessary logic. At the level of specific members of a federated collection of storage and querying containers, another similar API allows container specific logical functions to be registered for each container type that is itself registered as part of the federation. So for example, if one of the registered containers were a relational database system, it would not only register a 'kCreateTypeStorageFunc' function (which would be responsible for creating all storage tables etc. in that container that are necessary to handle the types defined in the ontology given) but also a variety of other functions. The constants for some of the more relevant plug-ins at the container level are given above. For example, the 'kCheckFieldType' plug-in could be called by the environment in order to determine which container in the federation will be responsible for the storage and retrieval of any given field in the type hierarchy. If we assume a very simple federation consisting of just two containers, a relational database, and an inverted text search engine, then we could imagine that the implementation of the 'kCheckFieldType' function for these two would be something like that given below:

```
// Inverted file text engine:

Boolean DTX_CheckFieldType     (                              // Field belongs
to 'TEXT" ?
                  ET_TypeID            aTypeID,               // I: Type ID
                  charPtr              fieldname              // I: Field name
                               )                              // R: Error code
(0 = no error)
{
   ET_TypeID           fType,baseType;
   int32               rType;
   Boolean             ret;

   fType = TM_GetFieldTypeID(NULL, aTypeID, fieldName);
   ret = NO;
   if ( TM_TypeIsReference(NULL, fType, &rType, &baseType) && baseType ==
kInt8Type &&
            (rType == kPointerRef || rType == kHandleRef || rType ==
kRelativeRef) )
      ret = YES;
   return ret;
}

// Relational database:

Boolean DSQ_CheckFieldType    (                          // Field belongs
to 'RDBM' ?
               ET_TypeID           aTypeID,              // I: Type ID
               charPtr             fieldname             // I: Field name
                              )                          // R: Error code
(0 = no error)
{
   ET_TypeID            fType, baseT;
   int32                refT;
   Boolean              ret;

   fType = TM_GetFieldTypeID(NULL, aTypeID, fieldname);
   ref = TM_TypeIsReference(NULL,fType,&refT,&baseT);
   ret = NO;
   if ( ref && refT == kPersistentRef )                  // We'll handle
pers. Refs.
      ret = YES;
   else if ( !ref && (                                   // We do:
      TM_IsTypeDescendant(NULL, fType, kInt8Type) ||     // char arrays,
      fType == TM_GetTypeID(NULL, "Date") ||             // Dates,
      TM_IsTypeDescendant(NULL,fType,kIntegerNumbersType)   ||     //
Integers and
      TM_IsTypeDescendant(NULL,fType,kRealNumbersType) ) )         //
Floating point #'s
      ret = YES;
   return ret;
}
```

As the pseudo-code above illustrates, in this particular federation, the inverted text engine lays claim to all fields that are references (normally '@') to character strings (but not fixed sized arrays of char) while the relational container lays claim to pretty much everything else including fixed (i.e., small sized) character arrays. This is just one possible division of responsibility is such a federation, and many others are possible. Other containers that may be members of such federations include video servers, image servers, map engines, etc. and thus a much more complex division of labor between the various fields of any given type will occur in practice. This ability to abstract away the various containers that form part of the persistent storage federation, while unifying and automating access to them, is a key benefit of the system of this invention.

Returning to the specifics of an RDBMS federation member, the logic associated with the 'kCreateTypeStorageFunc' plug-in for such a container (assuming an SQL database engine such as Oracle) might look similar to that given below:

```
static EngErr DSQ_CreateTypeStorage(                         // Build SQL
tables
                ET_TypeID             theType                // I: The type
                                )                            // R: Error Code
(0 = no error)
{
   char         sqlStatement[256], filter[256];

   err = DSQ_CruiseTypeHierarchy(theType,DSQ_CreateTypeTable);
   sprintf(filter,                                           // does linkage
table exist?
           "owner=(select username from all_users where user_id=uid) and "
           "table_name='LINKAGE_TABLES$'");

   if (#records found("all_tables", filter))                 // If not, then
create it!
   {
      sprintf(sqlStatement, "create table LINKAGE_TABLES$
          (DYN_NAME varchar2(50),ACT_NAME varchar2(50)) tablespace data");
      err = SQL_ExecuteStatement(0, sqlStatement, NULL, 0, NULL);
   }
   err = DSQ_CruiseTypeHierarchy(theType, DSQ_CreateLinkageTables);
   ... any other logic required
   return (err);
}
```

In this example, the function DSQ_CruiseTypeHierarchy() simply recursively walks the type hierarchy beginning with the type given down and calls the function specified. The function DSQ_CreateTypeTable() simply translates the name of the type (obtained from TM_GetTypeName) into the corresponding Oracle table name (possibly after adjusting the name to comply with constraints on Oracle table names) and then loops through all of the fields in the type determining if they belong to the RDBMS container and if so generates the corresponding table for the field (again after possible name adjustment). The function DSQ_CreateLinkageTables() creates anonymous linkage tables (based on field names involved) to handle the case where a field of the type is a collection reference, and the reference is to a field in another type that is also a collection reference echoing back to the original field. After this function has been run for all types in the ontology, it is clear that the external relational database now contains all tables and linkage tables necessary to implement any storage, retrieval and querying that may be implied by the ontology. Other registered plug-in functions for the RDBMS container such as query functions can utilize knowledge of the types hierarchy in combination with knowledge of the algorithm used by DSQ_CreateTypeStorage(), such as knowledge of the name adjustment strategy, to reference and query any information automatically based on type.

Note that some of the reference fields in the example above do not contain a '><' operator which implies that the ontology definer does not wish to have the necessary linking tables appear in the ontology. An example of such a field (as set forth above) is "Category @@membership". This field can be used to create an anonymous linkage table based on the type being referenced and the field name doing the referencing (after name adjustment). The linkage table would contain two references giving the type and ID of the objects being linked. When querying such an anonymous table, the plug-ins can deduce its existence entirely from the type information (and knowledge of the table creation algorithm) and thus the same querying power can be obtained even without the explicit definition of the linking table (as in the example above). Queries from the C* level are not possible directly on the fields of such a linkage table because it does not appear in the ontology, however, this technique is preferably used when such queries would not necessarily make sense.

By using this simple expedient, a system is provided in which external RDBMS storage is created automatically from the ontology itself, and for which subsequent access and querying can be handled automatically based on knowledge of the type hierarchy. This has effectively eliminated the need for a SQL database administrator or database programming staff. Since the same approach can be adopted for every container that is a member of the federation, these same capabilities can be accomplished simultaneously for all containers in the federation. As a result, the creator of a system based on this technology can effectively ignore the whole database issue once the necessary container plug-ins have been defined and registered. This is an incredibly powerful capability, and allows the system to adapt in an automated manner to changes in ontology without the need to consider database impact, thus greatly increasing system flexibility and robustness to change. Indeed, whole new systems based on this technology can be created from scratch in a matter of hours, a capability has been up until now unheard of. Various other plug-in functions may also be implemented, which can be readily deduced from this description.

The process of assigning (or determining) the unique ID associated with instantiating a persistent reference resulting from mining a datum from an external source (invoked via the $UniqueID script as further described in the Collections Patent) deserves further examination since it is highly dependant on the type of the data involved and because it further illustrates the systems ability to deal with such real-world quirks. In the simple federation described above, the implementation of the $UniqueID script for Datum (from which all other types will by default inherit) might be similar to that given below:

```
static EngErr PTS_AssignUniqueID(                    // $UniqueID script
registered with Datum
                  ET_TypeDBHdl        aTypeDBHdl, // I:Type DB handle (NULL
to default)
                  ET_TypeID           typeID,     // I:Type ID
                  charPtr             fieldName,  // I:Field name/path (else
NULL)
                  charPtr             action,     // I:The script action
being invoked
                  charPtr             script,     // I:The script text
                  anonPtr             dataPtr,    // I:Type data pointer
                  ET_CollectionHdl    aCollection,// I:The collection handle
                  ET_Offset           offset,     // I:Collection element
reference
                  int32               options,    // I:Various logical
options
```

```
                ET_TypeID               fromWho,      // I:Type ID, 0 for field
or unknown
                va_list                 ap            // I:va_list for additional
parameters
                            )                         // R:0 for success, else
error #
{
   ET_UniqueID       uniqueID;

   TC_GetUniqueID(aCollection,0,offset,&uniqueID);
   TC_GetCStringFieldValue(aCollection,0,0,offset,name,sizeof(name),"name")
;
   elemTypeID = TC_GetTypeID(aCollection,0,offset);
   TM_BreakUniqueID(uniqueID,&localID,&sys);
   if ( localID ) return 0;                          // we've already got an
ID,we're done!
   scrubbedStrPtr = mangle name according to SQL name mangling algorithm
   force scrubbedStrPtr to upper case
   sprintf(filterText, kStartQueryBlock kRelationalDB ":upper(name) = '%s'"
               kEndQueryBlock, scrubbedStrPtr); // Create the filter
criteria
   hitList = construct hit list of matches
   count = # hits in hitList;                        // how many hits did we get
   // Should issue a warning or dialog if more than one hit here
   if (hitList && hitList[0]._id)
   {
      uniqueID = TM_MakeUniqueID(hitList[0]._id,hitList[0]._system);
      existingElemTypeID = hitList[0]._type;
      exists = TRUE;
   }
   if (!uniqueID.id)
      uniqueID = TM_MakeUniqueID(DB_GetNextLocalUniqueID(),0);
   if (!TC_HasDirtyFlags(aCollection, 0, 0, offset))
      call TC_EstablishEmptyDirtyState(aCollection,0,0,offset,NO) )
   TC_SetUniqueID(aCollection,0,offset,uniqueID);// set the id
   return err;
}
```

This is a simple algorithm and merely queries the external RDBMS to determine if an item of the same name already exists and if so uses it, otherwise it creates a new ID and uses that. Suppose that the item involved is of type "Place". In this case, it would be helpful to be more careful when determining the unique ID because place names (such as cities) can be repeated all over the world (indeed there may be multiple cities or towns with the same within any given country). In this case, a more specific $UniqueID script could be registered with the type Place (the ancestral type of all places such as cities, towns, villages etc.) that might appear more like the algorithm given below:

```
static EngErr PTS_AssignPlaceUniqueID(              // $UniqueID script
registered with Place
                ET_TypeDBHdl        aTypeDBHdl,  // I:Type DB handle (NULL
to default)
                ET_TypeID           typeID,      // I:Type ID
                charPtr             fieldName,   // I:Field name/path (else
NULL)
                charPtr             action,      // I:The script action
being invoked
                charPtr             script,      // I:The script text
                anonPtr             dataPtr,     // I:Type data pointer
                ET_CollectionHdl    aCollection,// I:The collection handle
                ET_Offset           offset,      // I:Collection element
reference
                int32               options,     // I:Various logical
options
                ET_TypeID           fromWho,     // I:Type ID, 0 for field
or unknown
                va_list             ap           // I:va_list for additional
parameters
                            )                    // R:0 for success, else
error #
{
  ET_UniqueID         uniqueID;

  TC_GetUniqueID(aCollection,0,offset,&uniqueID);
  TC_GetCStringFieldValue(aCollection,0,0,offset,name,sizeof(name),"name")
;
  TC_GetCStringFieldValue(aCollection,0,0,offset,thisPlace,128,"placeType"
);
  TC_GetFieldValue(aCollection,0,0,offset,&thisLon,"location.longitude");
  TC_GetFieldValue(aCollection,0,0,offset,&thisLat,"location.latitude");
```

```
   elemTypeID = TC_GetTypeID(aCollection,0,offset);
   pT = TM_IsTypeProxy(elemTypeID);
   if ( pT ) elemTypeID = pT;

   TM_BreakUniqueID(uniqueID,&localID,NULL);
   if ( localID ) return 0;                    // we've already got an
ID,we're done!

   scrubbedStrPtr = mangle name according to SQL name mangling algorithm
   force scrubbedStrPtr to upper case
   sprintf(filterText, kStartQueryBlock kRelationalDB ":upper(name) = '%s'"
               kEndQueryBlock, scrubbedStrPtr);
   sprintf(fieldList,"placeType,location,country");
   tmpCollection = fetch all matching items to a collection
   TC_Count(tmpCollection,kValuedNodesOnly,rootElem,&count);
   // if we got one or more we need further study to see if it is in fact
this place
   // a place is unique if the place type, latitude and longitude are the
same
   placeTypeId = TM_KeyTypeToTypeID('PLCE',NULL);
   pplaceTypeId = TM_KeyTypeToTypeID('POPP',NULL);
   if (count)
   {
      anElem =0;
      while (tmpCollection && TC_Visit(tmpCollection,kRecursiveOperation +
                                  kValuedNodesOnly,0,&anElem,false))
      {
         if ( TM_TypesAreCompatible(NULL, TC_GetTypeID( tmpCollection, 0,
anElem)
         ,pplaceTypeId) &&
TM_TypesAreCompatible(NULL,elemTypeID,pplaceTypeId) )
         {                                      // both populated places,
check country
            TC_GetFieldValue(tmpCollection,0,0,anElem,&prf1,"country");
            TC_GetFieldValue(aCollection,0,0,offset,&prf2,"country");
            if (strcmp(prf1.name,prf2.name) )   // different country!
               continue;

   TC_GetCStringFieldValue(tmpCollection,0,0,anElem,&placeType,128,"placeTy
pe");
            if (!strcmp(thisPlace,placeType))   // same type
            {
               if (
TC_IsFieldEmpty(tmpCollection,0,0,anElem,"location.longitude") )
               {                                // this is the same place!
                  TC_GetUniqueID(tmpCollection,0,anElem,&uniqueID);
                  TM_BreakUniqueID(uniqueID,&localID,NULL);
                  existingElemTypeID =
TC_GetTypeID(tmpCollection,0,anElem);
                  exists = (existingElemTypeID != 0);
                  break;
               } else
               {
                  TC_GetFieldValue(tmpCollection, 0, 0, anElem, &longitude,
                                      "location.longitude");
                  if (ABS(thisLon - longitude) < 0.01)
                  {                             // at similar longitude
                     TC_GetFieldValue(tmpCollection, 0,0, anElem,
&latitude,
                                           "location.latitude");
                     if (ABS(thisLat - latitude) < 0.01)
                     {                          // and similar latitude!
                        TC_GetUniqueID(tmpCollection,0,anElem,&uniqueID);
                        TM_BreakUniqueID(uniqueID,&localID,NULL);
                        existingElemTypeID =
TC_GetTypeID(tmpCollection,0,anElem);
                        exists = (existingElemTypeID != 0);
                        break;
                     }
                  }
               }
            }
         }
      }
   }

   if ( !localID )
     uniqueID = TM_MakeUniqueID(DB_GetNextLocalUniqueID(),0);
   else
     uniqueID = TM_MakeUniqueID(localID,0);
  if (!TC_HasDirtyFlags(aCollection, 0, 0, offset))
     call TC_EstablishEmptyDirtyState(aCollection,0,0,offset,NO) )
   TC_SetUniqueID(aCollection,0,offset,uniqueID);// set the id
   return err;
}
```

This more sophisticated algorithm for determining place unique IDs attempts to compare the country fields of the Place with known places of the same name. If this does not distinguish the places, the algorithm then compares the place type, latitude and longitude, to further discriminate. Obviously many other strategies are possible and completely customizable within this framework and this example is provided for illustration purposes only. The algorithm for a person name, for example, would be completely different, perhaps based on age, address, employer and many other factors.

It is clear from the discussion above that a query-building interface can be constructed that through knowledge of the types hierarchy (ontology) alone, together with registration of the necessary plug-ins by the various containers, can generate the UI portions necessary to express the queries that are supported by that plug-in. A generic query-building interface, therefore, need only list the fields of the type selected for query and, once a given field is chosen as part of a query, it can display the UI necessary to specify the query. Thereafter, using plug-in functions, the query-building interface can generate the necessary query in the native language of the container involved for that field.

Referring now to Figure 2, a sample query-building user interface (UI) is shown. In this sample, the user is in the process of choosing the ontological type that he wishes to query. Note that the top few levels of one possible ontological hierarchy 210, 215, 220 are visible in the menus as the user makes his selection. A sample ontology is discussed in more detail below. The UI shown is one of many possibly querying interfaces and indeed is not that used in the preferred embodiment but has been chosen because it clearly illustrates the connections between containers and queries.

Referring now to Figure 3, a sample user interface providing access to the fields within the type "country" is shown. Having selected Country from the query-building UI illustrated in Figure 2, the user may then chose any of the fields of the type country 310 on which he wishes to query. In this example, the user has picked the field 'dateEntered' 320 which is a field that was inherited by Country from the base persistent type Datum. Once the field 320 has been selected, the querying interface can determine which member of the container federation is responsible for handling that field (not shown). Through registered plug-in functions, the querying language can determine the querying operations supported for that type. In this case, since the field is a date (which, in this example, is handled by the RDBMS container), the querying environment can determine that the available query operations 330 are those appropriate to a date.

Referring now to Figure 4, a sample user interface providing access to a free format text field within the type "country" is shown. In this figure, the user has chosen a field supported by the inverted text file container. Specifically, the field "notes.sourceNotes" has been chosen (which again is inherited from Datum) and thus the available querying operators 410 (as registered by the text container) are those that are more appropriate to querying a free format text field.

Referring now to Figure 5, a sample user interface providing access to a fixed sized text field within the type "country" is shown. In this figure, the user has chosen the field "geography.landAreaUnits" 510, which is a fixed sized text field of Country. Again, in the above illustration, this field is supported by the RDBMS container so the UI displays the querying operations 520 normally associated with text queries in a relational database.

The above discussion illustrated how container specific storage could be created from the ontology, how to query and retrieve data from individual containers in the federation, and how the user interface and the queries themselves can be generated directly from the ontology specification without requiring custom code (other than an application independent set of container plug-ins). The other aspects necessary to create a completely abstracted federated container environment relate to three issues: 1) how to distribute queries between the containers, 2) how to determine what queries are possible, and 3) how to reassemble query results returned from individual containers back into a complete record within a collection as defined by the ontology. The portion of the system of this invention that relates to defining individual containers, the querying languages that are native to them, and how to construct (both in UI terms and in functional terms) correct and meaningful queries to be sent to these containers, is hereinafter known as MitoQuest™. The portion of the system that relates to distributing (federating) queries to various containers and combining the results from those containers into a single unified whole, is hereinafter known as MitoPlex™. The federated querying system of this invention thus adopts a two-layer approach: the lower layer (MitoQuest™) relates to container specific querying, the upper layer (MitoPlex™) relates to distributing queries between containers and re-combining the results returned by them. Each will be described further below (in addition to the patent application referenced herein).

Each container, as a result of a container specify query, constructs and returns a hit-list of results that indicate exactly which items match the container specific query given. Hit lists are zero terminated lists that, in this example, are constructed from the type ET_Hit, which is defined as follows:

```
typedef struct ET_Hit                          // list of query hits returned by a server
{
   OSType          _system;                      // system tag
   unsInt64        _id;                          // local unique item ID
   ET_TypeID       _type;                        // type ID
   int32           _relevance;                      // relevance value 0..100
} ET_Hit;
```

As can be seen, an individual hit specifies not only the globally unique ID of the item that matched, but also the specific type involved and the relevance of the hit to the query. The specific type involved may be a descendant of the type queried since any query applied to a type is automatically applied to all its descendants since the descendants "inherit" every field of the type specified and thus can support the query given. In this embodiment, relevance is encoded as an integer number between 0 and 100 (i.e., a percentage) and its computation is a container specific matter. For example, this could be calculated by plug-in functions within the server(s) associated with the container. It should be noted that the type ET_Hit is also the parent type of all proxy types (as further discussed in the Types Patent) meaning that all proxy types contain sufficient information to obtain the full set of item data if required.

When constructing a multi-container query in MitoPlex™, the individual results (hit lists) are combined and re-assembled via the standard logical operators as follows:

AND - For a hit to be valid, it must occur in the hit list for the container specific query occurring before the AND operator and also in the hit list for the container specific query that follows the AND.

OR – For a hit to be valid, it must occur in either the hit list before the operator, or the one after the operator (or both).

AND THEN – This operator has the same net effect as the AND operator but the hit-list from before the operator is passed to the container executing the query that follows the operator along with the query itself. This allows the second container to locally perform any pruning implied by the hit list passed before returning its results. This operator therefore allows control over the order of execution of queries and allows explicit optimization of performance based on anticipated results. For example if one specified a mixed container query of the form "[RDBMS:date is today] AND THEN [TEXT:text contains "military"]" it is clear that the final query can be performed far quicker than the effect of performing the two queries separately and then recombining the results since the first query pre-prunes the results to only those occurring on a single day and since the system may contain millions of distinct items where the text contains "military". For obvious reasons, this approach is considerably more efficient.

AND {THEN} NOT – This operator implies that to remain valid, a hit must occur in the hit-list for the query specified before the operator but not in the hit-list for the query after the operator.

Additional logical operators allow one to specify the maximum number of hits to be returned, the required relevance for a hit to be considered, and many other parameters could also be formulated. As can be seen, the basic operations involved in the query combination process involve logical pruning operations between hit lists resulting from MitoQuest™ queries. Some of the functions provided to support these processes may be exported via a public API as follows:

```
Boolean DB_NextMatchInHitList (                  // Obtain the next match in
a hit list
              ET_Hit*           aMatchValue,     // I:Hit value to match
              ET_HitList        *aHitList,       // IO:Pointer into hit list
              int32             options          // I: options as for
DB_PruneHitList()
```

```
                                 );                // R:TRUE if match
found,else FALSE

Boolean DB_BelongsInHitList      (                 // Should hit be added to a
hit list?
               ET_Hit*              aHit,          // I:Candidate hit
               ET_HitList           aPruneList,    // I:Pruning hit list, zero
ID term.
               int32                options        // I:pruning options word
                                 );                // R:TRUE to add hit, FALSE
otherwise

ET_HitList DB_PruneHitList       (                 // prunes two hit lists
               ET_HitList           aHitList,      // I:Input hit list, zero
ID terminated
               ET_HitList           aPruneList,    // I:Pruning hit list, zero
ID term.
               int32                options,       // I:pruning options word
               int32                maxHits        // I:Maximum # hits to
return (or 0)
                                 );                // R:Resultant hit list, 0
ID term.
```

In the code above, the function **DB_NextMatchInHitList ()** would return the next match according to specified sorting criteria within the hit list given. The matching options are identical to those for DB_PruneHitList(). The function **DB_BelongsInHitList()** can be used to determine if a given candidate hit should be added to a hit list being built up according to the specified pruning options. This function may be used in cases where the search engine returns partial hit sets in order to avoid creating unnecessarily large hit lists only to have them later pruned. The function **DB_PruneHitList()** can be used to prune/combine two hit lists according to the specified pruning options. Note that by exchanging the list that is passed as the first parameter and the list that is passed as the second parameter, it is possible to obtain all possible behaviors implied by legal combinations of the MitoPlex™ AND, OR, and NOT operators. Either or both input hit lists may be NULL which means that this routine can be used to simply limit the maximum number of hits in a hit list or alternatively to simply sort it. In the preferred embodiment, the following pruning options are provided:

kLimitToPruneList - limit returned hits to those in prune list (same as MitoPlex™ AND)

kExclusiveOfPruneList - remove prune list from 'hits' found (same as MitoPlex™ AND NOT)

kCombineWithPruneList - add the two hit lists together (default - same as MitoPlex™ OR)

The following options can be used to control sorting of the resultant hit list:

kSortByTypeID	-- sort resultant hit list by type ID
kSortByUniqueID	-- sort resultant hit list by unique ID
kSortByRelevance	-- sort resultant hit list by relevance
kSortInIncreasingOrder	-- Sort in increasing order

In addition to performing these logical operations on hit lists, MitoPlex™ supports the specification of registered named MitoQuest™ functions in place of explicit MitoQuest™ queries. For example, if the container on one side of an operator indicates that it can execute the named function on the other side, then the MitoPlex™ layer, instead of separately launching the named function and then combining results, can pass it to the container involved in the other query so that it may be evaluated locally. The use of these 'server-based' multi-container queries is extremely useful in tuning system performance. In the preferred embodiment of the system based on this invention, virtually all containers can locally support interpretation of any query designed for every other container (since they are all implemented on the same substrate) and thus all queries can be executed in parallel with maximum efficiency and with pruning occurring in-line within the container query process. This approach completely eliminates any overhead from the federation process. Further details of this technique are discussed in related patent applications that have been incorporated herein.

It is clear from the discussion above that the distribution of compound multi-container queries to the members of the container federation is a relatively simple process of identifying the containers involved and launching each of the queries in parallel to the server(s) that will execute it. Another optimization approach taken by the MitoPlex™ layer is to identify whether two distinct MitoQuest™ queries involved in a full MitoPlex™ query relate to the same container. In such a case, the system identifies the logic connecting the results from each of these queries (via the AND, OR, NOT etc. operators that connect them) and then attempts to re-formulate the query into another form that allows the logical combinations to instead be performed at each container. In the preferred embodiment, the system performs this step by combining the separate queries for that container into a single larger query combined by a container supplied logical operator. The hit-list combination logic in the MitoPlex™ layer is then altered to reflect the logical re-arrangements that have occurred. Once again, all this behavior is possible by abstract logic in the MitoPlex™ layer that has no specific dependency on any given registered container but is simply able to perform these manipulations by virtue of the plug-in functions registered for each container. These registered plug-in functions inform the MitoPlex™ and MitoQuest™ layers what functionality the container can support and how to invoke it. This approach is therefore completely open-ended and customizable to any set of containers and the functionality they support. Examples of other container functionality might be an image server that supports such querying behaviors as 'looks like', a sound/speech server with querying operations such as 'sounds like', a map server with standard GIS operations, etc. All of these can be integrated and queried in a coordinated manner through the system described herein.

The next issue to address is the manner in which the present invention auto-generates and handles the user interface necessary to display and interact with the information defined in the ontology. At the lowest level, all compound structures eventually resolve into a set of simple building-block types that are supported by the underlying machine architecture. The same is true of any type defined as part of an ontology and so the first requirement for auto-generating user interface based on ontological specifications is a GUI framework with a set of 'controls' that can be used to represent the various low level building blocks. This is not difficult to achieve with any modern GUI framework. The following images and descriptive text illustrate just one possible set of such basic building blocks and how they map to the low level type utilized within the ontology:

Referring now to Figure 6A, an example of how a short text field or numeric field (such as those handled by the RDBMS container described above) might be displayed in a control group.

Referring now to Figure 6B, one method for displaying a date in a control group is shown. In this Figure, the date is actually being shown in a control that is capable of displaying dates in multiple calendar systems. For example, the circle shown on the control could be displayed in yellow to indicate the current calendar is Gregorian. Referring now to Figure 6C, an example of an Islamic Hijjrah calendar being displayed is provided. The UI layout can be chosen to include the calendar display option, for example.

Referring now to Figure 7A, the illustrated control group is an example of how one might display and interact with a persistent reference field ('#'). The text portion 705 of the grouping displays the name field of the reference, in this case 'InsuregencyAndTerrorism', while the list icon 710 allows the user to pop up a menu of the available values (see the 'members' field discussion under ET_PersistentRef above), and the jagged arrow icon 715 allows the user to immediately navigate to (hyperlink to) the item being referenced.

Referring now to Figure 7B, 7B provides an example of one way that a collection reference field ('@@') might be displayed in an auto-generated user interface. In this case the field involved is the 'related' field within the notes field of Datum. Note also that the collection in this case is hierarchical and that the data has been organized and can be navigated according to the ontology.

Referring now to Figure 8, one possible method for displaying variable sized text fields (referenced via the char @ construct) is shown. Note that in this example, automatic UI hyperlink generation has been turned on and thus any known item within the text (in this case the names of the countries) is automatically hyperlinked and can be used for navigation simply by clicking on it (illustrated as an underline). This hyperlinking capability will be discussed further in later patents but the display for that capability may be implemented in any number of ways, including the manner in which hyperlinks are displayed by web browsers.

Referring now to Figure 9, this figure illustrates the manner in which an image reference (Picture @picture) field could be displayed in an auto-generated user interface.

Many other basic building blocks are possible and each can of course be registered with the system via plug-ins in a manner very similar to that described above. In all cases, the human-readable label associated with the control group is generated automatically from the field name with which the control group is associated by use of the function

TM_CleanFieldName() described in the Types Patent. Because the system code that is generating and handling the user interface in this manner has full knowledge of the type being displayed and can access the data associated with all fields within using the APIs described previously, it is clear how it is also possible to automatically generate a user interface that is capable of displaying and allowing data entry of all types and fields defined in the ontology. The only drawback is the fact that user interfaces laid out in this manner may not always look 'professional' because more information is required in order to group and arrange the layout of the various elements in a way that makes sense to the user and is organized logically. The system of this invention overcomes this limitation by extracting the necessary additional information from the ontological type definition itself. To illustrate this behavior, a listing is provided in Appendix A that gives the pseudo-code ontological type definition for the type Country (which inherits from Entity and thereby from Datum described above) in the example ontology.

As can be seem from the listing above, the ontology creator has chosen to break down the many fields of information available for a country into a set of introductory fields followed by number of top-level sub-structures as follows:

geography	- Information relating to the country's geography
people	- Information relating to the country's people
government	- Information relating to the country's government
economy	- Information about the country's economy
communications	- Information on communications capabilities
transport	- Transport related information
military	- Information about the country's military forces
medical	- Medical information
education	- Education related information
issues	- Current and past issues for the country involved

Because the code that generates the UI has access to this information, it can match the logical grouping made in the ontology.

Referring now to Figure 10, a sample screen shot of one possible display of the Country record in the same UI layout theme described above (most data omitted) is provided. In the illustrated layout the first page of the country display shows the initial fields given for country in addition to the basic fields inherited from the outermost level of the Datum definition. The user is in the process of pulling down the 'page' navigation menu 1020 which has been dynamically built to match the ontology definition for Country given above. In addition, this menu contains entries 1010 for the notes sub-field within Datum (the ancestral type) as well as entries for the fields 1030 that country inherits from its other ancestral types. In the first page, the UI layout algorithm in this example has organized the fields as two columns in order to make best use of the space available given the fields to be displayed. Since UI layout is registered with the environment, it is possible to have many different layout strategies and appearances (known as themes) and these things are configurable for each user according to user preferences.

Referring now to Figure 11, a sample embodiment of the geography page within Country is shown. Presumably, the user has reached this page using the page navigation menu 1020 described above. In this case, the UI does not have sufficient space to display all fields of geography on a single page, so for this theme it has chosen to provide numbered page navigation buttons 1110, 11120, 1130 to allow the user to select the remaining portions of the geography structure content. Once again, different themes can use different strategies to handle this issue. The theme actually being shown in this example is a Macintosh OS-9 appearance and the layout algorithms associated with this theme are relatively primitive compared to others.

Referring now to Figure 12, a sample embodiment of the second sub-page of the geography page within country is shown. As shown, the natural resources collection field 1210 is displayed as a navigable list within which the user may immediately navigate to the item displayed simply by double-clicking on the relevant list row. More advanced themes in the system of this invention take additional measures to make better use of the available space and to improve the appearance of the user interface. For example, the size of the fields used to display variable sized text may be adjusted so that the fields are just large enough to hold the amount of text present for any given record. This avoids the large areas of white space that can be seen in Figure 12 and gives the appearance of a custom UI for each and every record displayed. As the window itself is resized, the UI layout is re-computed dynamically and a new appearance is established on-the-fly to make best use of the new window dimensions. Other tactics include varying the number of columns on each page depending on the information to be displayed, packing small numeric fields two to a column, use of disclosure tabs compact content and have it pop-up as the mouse moves over the tab concerned, etc. The possibilities are limited only by the imagination of the person registering the plug-ins. To achieve this flexibility, the UI layout essentially treats each field to be displayed as a variable sized rectangle that through a standard interface can negotiate to change size, move position or re-group itself within the UI. The code of the UI layout module allows all the UI components to compete for available UI space with the result being the final layout for a given ontological item. Clearly the matter of handling user entry into fields and its updating to persistent storage is relatively straightforward given the complete knowledge of the field context and the environment that is available in this system.

Referring now to Figure 13, an example of one part of a high-level ontology targeted at intelligence is shown. This ontology has been chosen to facilitate the extraction of meaning from world events; it does not necessarily correspond to any functional, physical or logical breakdown chosen for other purposes. This is only an example and in no way is such ontology mandated by the system of this invention. Indeed, the very ability of the system to dynamically adapt to any user-defined ontology is one of the key benefits of the present invention. The example is given only to put some of the concepts discussed previously in context, and to illustrate the power of the ontological approach in achieving data organization for the purposes of extracting meaning and knowledge. For simplicity, much detail has been omitted. The key to developing an efficient ontology is to categorize things according to the semantics associated with a given type. Computability must be independent of any concept of a 'database' and thus it is essential that these types automatically drive (and conceal) the structure of any relational or other databases used to contain the fields described. In this way, the types can be used by any and all code without direct reliance on or knowledge of a particular implementation.

Datum 1301 -- the ancestral type of all persistent storage.

Actor 1302 -- *actors* 1302 participate in *events 1303*, perform *actions 1305* on *stages 1304* and can be observed 1306.

Entity 1308-- Any 'unique' *actor1302* that has motives and/or behaviors, i.e., that is not passive

Country 1315-- a *country* 1315 is a unique kind of meta-*organization* with semantics of its own, in particular it defines the top level *stage* 1304 within which *events* 1303 occur (*stages* 1304 may of course be nested)

Organization 1316-- an *organization* 1316 (probably hierarchical)

Person 1317-- a *person1317*

SystemUser 1325-- a *person* 1317 who is a user of the system

Widget 1318-- an executable item (someone put it there for a purpose/motive!)

Object 1309-- A passive non-unique *actor* 1302, i.e., a thing with no inherent drives or motives

Equipment 1319-- An *object* 1309 that performs some useful function that can be described and which by so doing increases the range of *actions* 1305 available to an *Entity*1308.

Artifact 1320-- An *object* 1309 that has no significant utility, but is nonetheless of value for some purpose.

Stage 1304-- This is the platform or environment where *events* 1303 occur, often a physical location. *Stages* 1304 are more that just a place. The nature and history of a *stage* 1304 determines to a large extent the behavior and *actions* 1305 of the *Actors* 1302 within it. What makes sense in one *stage* 1304 may not make sense in another.

Action -- *actions* 1305 are the forces that *Actors* 1302 exert on each other during an *event* 1303. All *actions* 1305 act to move the *actor*(s) 1302 involved within a multi-dimensional space whose axes are the various *motivations* that an *Entity* 1308 can have (greed, power, etc.). By identifying the effect of a given type of *action* 1304 along these axes, and, by assigning *entities* 1308 'drives' along each motivational axis and strategies to achieve those drives, we can model behavior.

Observation -- an *observation* 1306 is a measurement of something about a *Datum* 1301, a set of data or an *event* 1303. *Observations* 1306 come from *sources* 1307.

General 1310-- A general *observation* 1301 not specifically tied to a given *datum* 1301.

Report 1321-- a *report* 1321 is a (partial) description from some perspective generally relating to an *Event*1303.

Story 1326-- a news story describing an *event* 1303.

Image 1327-- a still *image* of an *event* 1303.

Sound 1329-- a *sound* recording of an *event* 1303.

Video 1328-- a *video* of an *event* 1303.

Map 1330-- a *map* of an *event* 1303, *stage* 1304, or *entity* 1308.

Regarding 1311-- an *observation* regarding a particular *datum* 1301.

Note 1322-- a descriptive text note relating to the *datum* 1301.

CrossRef 1323-- an explicit one-way cross-reference indicating some kind of named 'relationship' exists between one *datum* 1301 and another, preferably also specifying 'weight' of the relationship.

Delta 1324-- an incremental change to all or part of a *datum* 1301, this is how the effect of the time axis is handled (a delta 1324 of time or change in time).

Relating 1312-- A bi-directional link connecting two or more data together with additional information relating to the link.

Source 1307-- A source is a logical *source* of *observations* 1306 or other Data.

Feed 1313-- Most *sources* 1307 in the system consist of Client/Server servers that are receiving one or more streams of *observations* 1306 of a given type, that is; a newswire server is a source that outputs *observations* 1306 of type *Story*. In the preferred embodiment, feed sources 1313 are set up and allowed to run on a continuous basis.

Query 1314-- sub-type of *source* 1307 that can be issued at any time, returning a collection of *observations* 1306 (or indeed any Datum 1301 derived type). The Query source type corresponds to one's normal interpretation of querying a database.

Event 1303-- An *event* is the interactions of a set of *actors* 1302 on a *stage* 1304. *Events* 1303 must be reconstructed or predicted from the *observations* 1306 that describe them. It is the ability to predict *events* 1303 and then to adjust *actions* 1305 based on *motives* (not shown) and strategies that characterizes an *entity* 1308. It is the purpose of an intelligence system to discover, analyze and predict the occurrence of events 1303 and to present those results to a decision maker in order that he can take Actions 1305. The Actions 1305 of the decision maker then become a 'feed' to the system allowing the model for his strategies to be refined and thus used to better find opportunities for the beneficial application of those strategies occurring in the data stream impinging on the system.

Once the system designer has identified the ontology that is appropriate to allow the system to understand and manipulate the information it is designed to access (in the example above – understanding world events), the next step is to identify what sources of information, published or already acquired, are available to populate the various types defined in the system ontology. From these sources and given the nature of the problem to be solved, the system designed can then define the various fields to be contained in the ontology and the logical relationships between them. This process is expressed through the C* ontology definition and the examples above illustrate how this is done. At the same time, awareness of the desired user interface should be considered when building an ontology via the C* specifications. The final step is to implement any ontology-specific scripts and annotations as described in the Collections Patent. Once all this is done, all that is necessary is to auto-generate all storage tables necessary for the system as described and then begin the process of mining the selected sources into the system.

Having mined the information (a very rapid process), the system designer is free to evolve this ontology as dictated by actual use and by the needs of the system users. Because such changes are automatically and instantaneously reflected throughout the system, the system is now free to rapidly evolve without any of the constraints implied by the Bermuda Triangle problem experienced in the prior art. This software environment can be rapidly changed and extended, predominantly without any need for code modification, according to requirements, and without the fear of introducing new coding errors and bugs in the process. Indeed system modification and extension in this manner is possible by relatively un-skilled (in software terms) customer staff themselves meaning that it no longer requires any involvement from the original system developer. Moreover, this sytem can, through the ontology, unify data from a wide variety of different and incompatible sources and databases into a single whole wherein the data is unified and searchable without consideration of source. These two capabilities have for years been the holy grail of all software development processes, but neither has been achieved—until now.

The foregoing description of the preferred embodiments of the invention has been presented for the purposes of illustration and description. For example, although described with respect to the C programming language, any programming language could be used to implement this invention. Additionally, the claimed system and method should not be limited to the particular API disclosed. The descriptions of the header structures should also not be limited to the embodiments described. While the sample pseudo code provides examples of the code that may be used, the plurality of implementations that could in fact be developed is nearly limitless. For these reasons, this description is not intended to be exhaustive or to limit the invention to the precise form disclosed. Many modifications and variations are possible in light of the above teaching. It is intended that the scope of the invention be limited not by this detailed description, but rather by the claims appended hereto.

APPENDIX A

```
typedef struct ET_DBInvokeRec
{
   int32          options;                           // invocation options
   OSType         dataType;                          // The data type
involved
   OSType         droneType;                         // drone type involved
(if any)
   int32          caller;                            // widget ID of original
caller
   ET_UniqueID    anItemID;                          // globally unique ID
   char           IPname[256];                       // optional identifying
name
   PicHandle      picH;                              // possible picture
handle
   char           action[32];                        // action requested by
invoker
   char           dataItemType[64];                  // data type name (if
!dataType)
   char           name[256];                         // data item/event name
   void*          aDBSdataIDptr;                     // pointer to DB data
record
   void*          dataPtr;                           // type specific
data/fields
   void**         dataHdl;                           // type specific
data/fields
   ET_Offset      element;                           // offset if collection
reference
} ET_DBInvokeRec;

typedef void (*ET_SymbolicFunc)(charPtr,void *,int32,int32);
typedef void (*ET_SymbolicReply)(charPtr,void *,int32,int32);

#define kGlobalSCOPE          -32768                 // Global scope
#define kViewLParentSCOPE     -3                     // Scope=view that
launched us
#define kViewPackSCOPE        -2                     // View pack scope
#define kLocalViewSCOPE       -1                     // Scope is the
enclosing view
#define kThisWidgetSCOPE      0                      // Scope is the current
widget
#define kWidgetParentSCOPE    1                      // Scope is this
widget's parent

Boolean OC_RegisterFunction   (                      // Register a symbolic
function
               int32               aWidgetID,        // I:ID of defining
widget
               charPtr             aFuncName,        // I:symbolic function
name
               charPtr             aFuncDesc,        // I:description of
function
               short               ancestorSpec,     // I:scope:See notes
               ET_SymbolicFunc     aFunctionPtr,     // I:Function address
```

```
                int32                   options             // I:various logical
options
                               );                           // R:TRUE for success

#define kFarFunction           0x00000001                   // Far function (i.e.,
not near)
#define kNearFunction          0x00000000                   // Near function (i.e.,
not far)
#define kDistinguishFuncPtrs   0x00000002                   // Allow multiple
registrations

Boolean OC_DeRegisterFunction  (                            // Remove fn. from
registry
                int32                   aWidgetID,          // I:ID of defining
widget
                charPtr                 aFuncName,          // I:symbolic fn. name
                short                   ancestorSpec,       // I:scope:see notes
                ET_SymbolicFunc         aMatchFuncAddr      // I:Matching fn.
address or NULL
                               );                           // R:TRUE for success

Boolean OC_DisableFunction     (                            // Disable symbolic fn.
                int32                   aWidgetID,          // I:defining widget ID
                charPtr                 aFuncName,          // I:symbolic fn. name
                short                   ancestorSpec        // I:scope:see notes
                               );                           // R:TRUE for success

Boolean OC_EnableFunction      (                            // Enable symbolic
function
                int32                   aWidgetID,          // I:defining widget ID
                charPtr                 aFuncName,          // I:symbolic fn. name
                short                   ancestorSpec        // I:scope:see notes
                               );                           // R:TRUE if state
changed

Boolean OC_FunctionIsDisabled  (                            // Is function disabled?
                int32                   aWidgetID,          // I:Defining widget ID
                charPtr                 aFuncName,          // I:Symbolic fn. name
                short                   ancestorSpec        // I:scope:see notes
                               );                           // R:TRUE if disabled

Boolean OC_DeRegisterAllFuncs  (                            // Remove all widget
registrations
                int32                   aWidgetID           // I:Widget ID (0 =
current)
                               );                           // R:TRUE for success

Boolean OC_CallSymbolicFunction(                            // call a symbolic
function
                charPtr                 aFuncName,          // I:symbolic fn. name
                void                    *aFuncParameter,    // I:fn. parameter (or
NULL)
                ET_SymbolicReply        aReplyFunc,         // I:reply fn. address
 (or NULL)
                int32                   aMatchWidgetID,     // I:Matching widget ID
or 0
```

```
                  ET_SymbolicFunc       aMatchFuncAddr,    // I:Matching fn. addr.
 (or NULL)
                  int32                 options            // I:Various logical
options
                                 );                        // R:TRUE for success


#define kRsrvdSymbOptsMask    0xFFF00000                   // reserved options mask
#define kNoParameterDelete    0x10000000                   // !delete parameter
after calls
#define kNoNameAndIdOverride  0x20000000                   // !name and ID override
#define kNoNameOverride       0x40000000                   // !name override (see
CS_Invoke)
#define kPreferNearBehavior   0x08000000                   // caller prefers a
near/modal
#define kNoGlobalSearch       0x04000000                   // !examine
globalregistry

#define kSymbParamTypeMask    0x00F00000                   // param type mask

#define kSymbParamTypeInvRec  0x00100000                   // parameter is
ET_IPInvokeRecPtr
#define kSymbParamTypeInteger 0x00200000                   // parameter is a &long
#define kSymbParamTypeString  0x00300000                   // parameter is a C
string
#define kSymbParamTypeHandle  0x00400000                   // parameter is typed
handle
#define kSymbParamTypeViewRef 0x00500000                   // parameter is a view
ref

int32 OC_CountSymbolicFunctions(                           // count functions
@scope
                int32                 aWidgetID,           // I:defining widget ID
                charPtr               aFuncName,           // I:symbolic function
name
                int32Hdl              *widgetIDs,          // O:zero term. widget
ID list
                Boolean               includeSupressed     // I:TRUE to include
disabled fns.
                               );                          // R:# widgets
registered

ET_SymbolicFunc OC_ResolveSymbolicFunction(                // Get function address
                int32                 *aWidgetID,          // IO:widget ID
                int32                 *aScopeID,           // O:ID of scope
widget(0 global)
                Boolean               *farFunction,        // O:TRUE/FALSE far/near
                charPtr               aFuncName,           // I:symbolic function
name
                int32                 aMatchWidgetID,      // I:Matching widget ID
(or 0)
                ET_SymbolicFunc       aMatchFuncAddr,      // I:Matching fn.
address or NULL
                Boolean               includeSupressed     // I:TRUE includes
disabled fns.
                               );                          // R:Function pointer or
NULL
```

A-3

```
EngErr OC_SetSymbolicFuncData (                                 // Set symb. fn. data
type
               int32                aWidgetID,                 // I:defining widget ID
               charPtr              aFuncName,                 // I:name of the
symbolic function
               Handle               aDataHandle,               // I:Handle for data to
be set
               OSType               aDataType,                 // I:The type of the
data to be set
               ET_SymbolicFunc      aMatchFuncAddr             // I:Matching fn.addr.
or NULL
                              );                               // R:0 for success, else
error#

EngErr OC_GetSymbolicFuncData (                                 // Get function type &
info
               int32                aWidgetID,                 // I:defining widget ID
               charPtr              aFuncName,                 // I:name of the
symbolic function
               Handle               *aDataHandle,              // O:Handle for attached
data
               OSType               *aDataType,                // O:The type of the
data
               int32                aMatchWidgetID,            // I:Matching widget ID
or 0
               ET_SymbolicFunc      aMatchFuncAddr             // I:Matching fn.addr.
(or NULL)
                              );                               // R:0 for success,else
Error #

EngErr OC_SetSymbolicFuncFlags(                                 // Set function flags
               int32                aWidgetID,                 // I:defining widget ID
               charPtr              aFuncName,                 // I:symbolic function
name
               int32                theFlags,                  // I:flags value
               int32                aMatchWidgetID,            // I:Matching widget ID
or 0
               ET_SymbolicFunc      aMatchFuncAddr             // I:Matching fn.addr.
(or NULL)
                              );                               // R:0 for success,else
error#

EngErr OC_GetSymbolicFuncFlags(                                 // Get function flags
word
               int32                aWidgetID,                 // I:defining widget ID
               charPtr              aFuncName,                 // I:symbolic function
name
               int32                *theFlags,                 // O:Flags word
               int32                aMatchWidgetID,            // I:Matching widget ID
or 0
               ET_SymbolicFunc      aMatchFuncAddr             // I:Matching fn.addr.
(or NULL)
                              );                               // R:0 for success,else
error#
```

A-4

```
charHdl OC_GetSymbolicFuncDesc(                             // Get function
description
                int32               aWidgetID,              // I:defining widget ID
                charPtr             aFuncName,              // I:symbolic function
name
                int32               aMatchWidgetID,         // I:Matching widget ID
or 0
                ET_SymbolicFunc     aMatchFuncAddr          // I:Matching fn.addr.
(or NULL)
                              );                            // R:descriptive text or
NULL

charHdl OC_ListSymbolicFunctions(                           // get alphabetized fn.
list
                int32               aWidgetID,              // I:defining widget ID
                short               ancestorSpec,           // I:scope:see notes
                int32               *count                  // O:count of functions
                              );                            // R:symbolic fn. list,
or NULL

Boolean OC_WidgetIDtoAncestorSpec(                          // widget ID to ancestor
Spec
                int32               ownWidgetID,            // I:ID of defining
widget
                int32               aWidgetID,              // I:ID of potential
ancestor
                short               *ancestorSpec           // O:corresponding
ancestor spec
                              );                            // R:TRUE if ancestor
found

int32 OC_AncestorSpecToWidgetID(                            // ancestor spec to
widget ID
                int32               aWidgetID,              // I:ID of defining
widget
                short               ancestorSpec            // I:Scope:see notes
                              );                            // R:Ancestor widget ID,
or 0

int32 OC_LowestCommonAncestor (                             // Find lowest shared
ancestor ID
                int32            aWidgetID1,                // I:ID of first widget
                int32            aWidgetID2                 // I:ID of second widget
                              );                            // R:widget ID or 0

Boolean DB_DefineHyperlinkDomain(                           // Define a hyperlink
domain
                OSType              aDataType,              // I:The data type for
the domain
                charPtr             aDomainName,            // I:C string for the
domain name
                charPtr             hyperlinkAction,        // I:Action to invoke
                int32               options                 // I:various logical
options
                              );                            // R:TRUE for success
```

A-5

```
#define kIsSystemDomain               0x00000001        // options - system
domain
#define kCaseInsensitiveDomain        0x00000002        // options - case
insensitive
#define kNoRecalcDomains              0x00000004        // options - suppress
recalculation

Boolean DB_UnDefineHyperlinkDomain(                     // Undefine a hyperlink
domain
               charPtr                aDomainName       // I:domain name
                              );                        // R:TRUE for success

EngErr DB_AddToDomainDictionary(                        // Add a new target to a
domain
               charPtr                aDomainName,      // I:Domain name C
string
               charPtr                aTargetName,      // I:Case insensitive
name
               ET_UniqueID            aUniqueID,        // I:A unique ID for
invoke
               int32                  options           // I:Various logical
options
                              );                        // R:Zero for success,
else error#

#define kNoSaveDomainToFile           0x0001            // options - !save
domain to file

EngErr DB_SubFromDomainDictionary(                      // Remove a target from
domain
               charPtr                aDomainName,      // I:Domain name C
string
               charPtr                aTargetName,      // I:Case insensitive
name
               int32                  options           // I:Various logical
options
                              );                        // R:0 for success, else
error#

void DB_NotifyHyperlinkChange (                         // Update UI to reflect
a change
               void
                              );                        // R:void

Boolean DB_IsHyperlinkTarget  (                         // Is string a hyperlink
target?
               OSType                 *aDataType,       // IO:See notes
               charPtr                aDomainName,      // IO:See notes
               charPtr                aTargetName,      // I:possible target
name
               charPtr                anAction,         // IO:If !NULL,
hyperlink action
               ET_UniqueIDPtr         aUniqueID,        // IO:If !NULL, holds
unique ID
               int32Ptr               numChars,         // IO:If !NULL, holds #
of chars
```

A-6

```
                int32Ptr                tokenSize,        // IO:If !NULL, holds
token size
                int32                   maxChar           // I:Maximum char# to
examine,0
                                );                        // R:TRUE if target
found

Boolean DB_HyperlinkToTarget    (                         // Hyperlink to a target
                OSType                  aDataType,        // I:Data type (if
known), 0 all
                charPtr                 aDomainName,      // I:The domain or NULL
for all
                charPtr                 aTargetName       // I:possible target
                                );                        // R:TRUE if hyperlink
occurred

OSType DB_IsKnownDomain         (                         // Is hyperlink domain
known?
                charPtr                 aDomainName,      // I:Domain name
                Boolean                 *isSysDomain,     // IO:TRUE if a system
domain
                Boolean                 *isLeafDomain,    // IO:TRUE if a leaf
domain
                Boolean                 *isAutoActivate,  // IO:TRUE if domain
auto-activates
                Boolean                 *isAutoCompact    // IO:TRUE if domain
compacts
                                );                        // R:domain data type,0
otherwise

ET_LexHdl DB_IsActiveDomain     (                         // Is hyperlink domain
active?
                charPtr                 aDomainName       // I:Domain name C
string
                                );                        // R:domain dictionary,
else NULL

Boolean DB_ActivateDomain       (                         // Activate a hyperlink
domain
                charPtr                 aDomainName,      // I:Domain name
(possibly partial)
                Boolean                 compact           // I:set TRUE to compact
                                );                        // R:TRUE for success

Boolean DB_DeActivateDomain     (                         // Deactivate hyperlink
domain
                charPtr                 aDomainName       // I:Domain name
(possibly partial)
                                );                        // R:TRUE for success

Boolean DB_GetDomainAction      (                         // Obtain the domain
'action'
                charPtr                 aDomainName,      // I:Domain name C
string
                charPtr                 hyperlinkAction   // O:Holds the domain
'action'
                                );                        // R:TRUE for success
```

A-7

```
Boolean DB_SetDomainAutoFlags (                          // Set/Clear domain
flags
               charPtr               aDomainName,        // I:Domain name C
string
               Boolean               autoActivate,       // I:TRUE to auto-
activate @start
               Boolean               autoCompact         // I:TRUE to compact
@activate
                            );                           // R:TRUE for success

typedef EngErr (*ET_DomainPopFunc)(OSType,charPtr,charPtr,long);

Boolean DB_SpecifyDomainPopulator(                       // Specify a domain
populator fn.
               charPtr               aDomainName,        // I:Domain name C
string
               ET_DomainPopFunc      aPopulatorFunc,     // I:The domain
populator fn.
               charPtr               populatorDesc       // I:Description of fn.
                            );                           // R:TRUE for success

EngErr DB_CallDomainPopulator (                          // Call the domain
'populator'
               charPtr               aDomainName,        // I:Domain name C
string
               long                  aParam              // I:pass custom
parameter
                            );                           // R:0 success, else
error #

Boolean DB_UseDefaultDomainPopulator(                    // use default domain
populator
               charPtr               aDomainName         // I:Domain name C
string
                            );                           // R:TRUE for success

Boolean DB_FindNextHyperlinkInText(                      // Find next hyperlink
in text
               OSType                aDataType,          // I:Data type, 0 = all
active
               charPtr               aDomainName,        // I:Domain name, NULL
for all
               charPtr               text,               // I:Text being scanned
               int32Ptr              context,            // IO:Context storage
location
               int32Ptr              startChar,          // IO:holds starting
char #
               int32Ptr              tokenSize,          // IO:holds # of chars
in name
               int32                 maxChar             // I:Maximum char#, 0
for all
                            );                           // R:TRUE if hyperlink
found

ET_LexHdl DB_ListKnownDomains (                          // Get Lex DB of known
domains
```

```
               Boolean                    systemDomains     // I:TRUE/FALSE =
system/user
                              );                            // R:Lex DB of domains,
else NULL

charHdl DB_ListActions        (                             // get list of all
invoker actions
               OSType                     aDataType,        // I:The data type (or
0)
               charPtr                    aDataItemType     // I:data type name,
NULL = all
                              );                            // R:action list, NULL
if error

Boolean DB_DataTypeToName     (                             // obtain the full data
type name
               OSType                     aDataType,        // I:Data type
               charPtr                    aBuffer           // IO:Contains full name
                              );                            // R:TRUE for success

OSType DB_NameToDataType      (                             // get data type from
[alt.] name
               charPtr                    aDataTypeString,  // I:Full data type/alt.
name
               Boolean                    noResolveAlts     // I:TRUE suppresses
alt. names
                              );                            // R:data type, else 0

void DB_OSTypeToString        (                             // Convert long to
OSType string
               long                       anOStype,         // I:A long to be
converted
               charPtr                    cp                // O:Buffer to contain
output
                              );                            // R:void

Boolean DB_OverrideForTypeAndItemExists(                    // overridden for type &
itemID?
               OSType                     aDataType,        // I:data type(0 use
aDataItemType)
               charPtr                    aDataItemType,    // I:data type string
(or NULL)
               charPtr                    action,           // I:Action (NULL =
"Display")
               ET_SymbolicFunc            *aFunction,       // O:function address on
exit
               int32                      *ancestorSpec,    // O:ancestor spec on
exit
               Handle                     *aFuncDataHandle, // O:symb. fn. data
Handle
               OSType                     *aFuncDataType,   // O:Fn. associated data
type
               int32                      *aFuncFlags,      // O:function flags word
               ET_UniqueID                anItemID          // I:The unique item ID
                              );                            // R:TRUE if override
exists
```

```
Boolean DB_OverrideForTypeAndItem(                    // Register type & ID
override
               OSType              aDataType,         // I:The data type or 0
               charPtr             aDataItemType,     // I:data type name or
NULL
               charPtr             action,            // I:Action (NULL =
"Display")
               short               ancestorSpec,      // I:scope:see notes
               ET_SymbolicFunc     aSymbFuncToCall,   // I:Symbolic function
to call
               ET_UniqueID         anItemID,          // I:The unique item ID
               Boolean             farFunction,       //
I:TRUE/FALSE'far'/'near' fn.
               Handle              aFuncDataHandle,   // I:Associated fn. data
handle
               OSType              aFuncDataType,     // I:Data type of
'aFuncDataHandle'
               int32               aFuncFlags         // I:Flags for symbolic
function
                            );                        // R:TRUE for success

Boolean DB_UndoOverrideForTypeAndItem(                // Undo type & ID
override
               OSType              aDataType,         // I:The data type or 0
               charPtr             aDataItemType,     // I:data type name, or
NULL
               charPtr             action,            // I:Action (NULL is
"Display")
               short            ancestorSpec,         // I:scope:see notes
               ET_UniqueID         anItemID           // I:unique item ID (0 =
all)
                            );                        // R:TRUE for success

Boolean DB_DisableOverrideForTypeAndItem(             // Disable by type,ID &
scope
               OSType              aDataType,         // I:The data type or 0
               charPtr             aDataItemType,     // I:Data type name or
NULL
               charPtr             action,            // I:Action (NULL is
"Display")
               short            ancestorSpec,         // I:scope:see notes
               ET_UniqueID         anItemID           // I:Unique item ID
(0=all)
                            );                        // R:TRUE for success

Boolean DB_EnableOverrideForTypeAndItem(              // Enable by type, ID
and scope
               OSType              aDataType,         // I:The data type or 0
               charPtr             aDataItemType,     // I:Data type name or
NULL
               charPtr             action,            // I:Action (NULL is
"Display")
               short               ancestorSpec,      // I:scope:see notes
               ET_UniqueID         anItemID           // I:The unique item ID
                            );                        // R:TRUE if disabled
```

```
Boolean DB_OverrideForType      (                              // Override by action &
type
                    OSType              aDataType,             // I:The data type or 0
                    charPtr             aDataItemType,         // I:Data type name or
NULL
                    charPtr             action,                // I:Action (NULL is
"Display")
                    short               ancestorSpec,          // I:scope:see notes
                    ET_SymbolicFunc     aSymbFuncToCall,       // I:Symbolic function
                    Boolean             farFunction,           // I:TRUE/FALSE far/near
fn.
                    Handle              aFuncDataHandle,       // I:Function data
handle
                    OSType              aFuncDataType,         // I:data handle type
                    int32               aFuncFlags             // I:Function flags
                                );                             // R:TRUE for success

Boolean DB_UndoOverrideForType(                                // undo override by
action & type
                    OSType              aDataType,             // I:Data type or 0
                    charPtr             aDataItemType,         // I:Data type name or
NULL
                    charPtr             action,                // I:Action (NULL is
"Display")
                    short               ancestorSpec           // I:scope:see notes
                                );                             // R:TRUE for success

Boolean DB_DisableOverrideForType(                             // Disable by type &
action
                    OSType              aDataType,             // I:The data type or 0
                    charPtr             aDataItemType,         // I:Data type name or
NULL
                    charPtr             action,                // I:Action (NULL is
"Display")
                    short               ancestorSpec           // I:scope:see notes
                                );                             // R:TRUE for success

Boolean DB_EnableOverrideForType(                              // Enb. override by type
& action
                    OSType              aDataType,             // I:The data type or 0
                    charPtr             aDataItemType,         // I:Data type name or
NULL
                    charPtr             action,                // I:Action (NULL is
"Display")
                    short               ancestorSpec           // I:scope:see notes
                                );                             // R:TRUE if disabled

Boolean DB_OverridesForTypeDisabled(                           // Overrides by type
disabled?
                    OSType              aDataType,             // I:Data type or 0
                    charPtr             aDataItemType,         // I:Data type name or
NULL
                    charPtr             action,                // I:The action (NULL is
"Display")
                    short               ancestorSpec           // I:scope (see notes)
                                );                             // R:TRUE if disabled
```

```
Boolean DB_OverridesForTypeAndItemDisabled(             // Type & ID overrides
disabled?
               OSType              aDataType,           // I:The data type or 0
               charPtr             aDataItemType,       // I:Data type name or
NULL
               charPtr             action,              // I:Action (NULL is
"Display")
               short               ancestorSpec,        // I:scope:see notes
               ET_UniqueID         anItemID             // I:The unique item ID
                              );                        // R:TRUE if disabled

Boolean DB_OverrideForTypeExists(                       // type overridden ?
               OSType              aDataType,           // I:The data type or 0
               charPtr             aDataItemType,       // I:Data type name or
NULL
               charPtr             action,              // I:Action (NULL is
"Display")
               ET_SymbolicFunc     *aFunction,          // O:function address on
exit
               int32               *ancestorSpec,       // O:ancestor spec on
exit
               Handle              *aFuncDataHandle,    // O:attached data
               OSType              *aFuncDataType,      // O:type of
'aFuncDataHandle'
               int32               *aFuncFlags          // O:function flags word
                              );                        // R:TRUE if override
exists

Boolean DB_DefineInvoker      (                         // define type Invoker
               OSType              aDataType,           // I:Data type
               charPtr             actionName,          // I:Action name
               ET_dbViewInvoker    anInvokerFn          // I:function to call
                              );                        // R:TRUE for success

ET_ViewHdl DB_Invoke          (                         // Invoke action handler
               OSType              aDataType,           // I:Data type
               charPtr             actionName,          // I:Action name or NULL
               ET_DBInvokeRecPtr   iR,                  // IO:The invoker record
               int32               options              // I:Various logical
options
                              );                        // R:View handle or NULL

ET_dbViewInvoker DB_DoesInvokerExist(                   // Does an invoker
exist?
               OSType              aDataType,           // I:Data type
               charPtr             actionName           // I:Action name
                              );                        // R:Invoker function
address

Boolean DB_UnDefineInvoker    (                         // remove existing
invoker fn.
               OSType              aDataType,           // I:Data type
               charPtr             actionName           // I:Action name
                              );                        // R:TRUE if invoker
removed
```

```
int32 PU_CursorToHyperlink    (                          // Cursor position to
hyperlink
               OSType                *aDataType,         // IO:data type if a
hyperlink
               charPtr               aDomainName,        // IO:domain name if
hyperlink
               charPtr               aTargetName,        // IO:target string if
hyperlink
               charPtr               anAction,           // IO:action if
hyperlink
               ET_UniqueIDPtr        aUniqueID,          // IO:unique ID if
hyperlink
               Boolean               selectHyperlink     // I:If TRUE, selects
hyperlink
                                     );                  // R:text hyperlink
index or 0

void PU_NotifyHyperlinkChange (                          // Update any UI for
hyperlinks
                     void
                                     );                  // R:void
```

60092186_2.DOC

CLAIMS

1) A system supporting synchronous and asynchronous, inter-thread function calls comprising:

   a threaded environment, wherein such threaded environment associates arbitrary data with one or more threads and includes a hierarchical nesting of thread contexts with one or more corresponding UI context relationships;
   one or more function registries, wherein such registries are associated with one or more contexts in the threaded environment; and
   an API, wherein such API is capable of invoking functions by name with the parameters supplied by a caller.

2) The system of claim 1, wherein the threaded environment associates function registries with one or more threads.

3) The system of claim 1, wherein each function registry includes an ancestral scope.

4) The system of claim 3, wherein the function registries further include a global registry has an ancestral scope that incorporates all other function registries.

5) The system of claim 1, wherein the function registries are capable of registering one or more functions by name.

6) The system of claim 5, wherein the function registries are capable of registering one or more functions at a given ancestral scope and associating such functions with one or more threads.

7) The system of claim 1, wherein the threaded environment is capable of passing events containing messages between threads.

8) The system of claim 1, wherein the threaded environmet is capable of retrieving threads based on a unique thread identifier.

9) The system of claim 6, wherein the API supports searches of the registered functions.

10) The system of claim 9, wherein the API supports searches of the registered functions in an order determined by the ancestral scope of such functions.

11) The system of claim 10, wherein the API supports execution of the registered function(s) located responsive to the search.

12) The system of claim 6, wherein the API associates arbitrary data and logical flags with registered functions.

13) The system of claim 11, wherein the API associates arbitrary data and logical flags with registered functions by direct calls to functions included in such API.

14) The system of claim 11, wherein the API associates arbitrary data and logical flags with registered functions in response to an event.

15) The system of claim 11, further comprising a reply system, wherein such reply system returns any results obtained from executing one or more registered functions to the calling context in a synchronous manner.

16) The system of claim 11, further comprising a reply system, wherein such reply system returns any results obtained from executing one or more registered functions to the calling context in a asynchronous manner.

17) The system of claim 11, wherein the API includes the ability to inhibit or enable execution of one more registered functions by ancestral scope.

18) The system of claim 1, further comprising one or more widgets.

19) The system of claim 18, wherein the widgets include one or more widgets nested into another widget.

20) The system of claim 19, wherein the widgets include one more widgets that can cause another widget to be executed.

21) The system of claim 19, wherein one widget is nested into another widget up to an arbitrary depth, wherein such depth does not to exceed 32767.

22) The system of claim 19, further comprising a User Interface for displaying data to a user.

23) The system of claim 22, wherein the User Interface includes one or more hyperlinks to a target.

24) The system of claim 23, wherein the hyperlinks target data stored within an ontological framework.

25) The system of claim 24, further comprising a display handler for displaying the target when the hyperlink is selected.

26) The system of claim 25, wherein the display handler is capable of calling one or more widgets when the hyperlink is selected.

27) The system of claim 22, further comprising system-wide and user-specific hyperlink domains.

28) The system of claim 27, wherein the system-wide and user-specific hyperlink domains are created using the lexical analyzer claimed in the Lexical Patent.

29) A method for providing a user-driven user interface to system data stored according to a system ontology, comprising the steps of:

registering one or more functions with a thread;

defining a scope for such registered functions;

searching for a set of named functions based on the scope assigned to such functions;

altering the characteristics of any text or text portion displayed within the user interface based on one more more hyperlink dictionary(s);

linking such text or text portion, as defined in the hyperlink dictionary(s), to system data stored using the system ontology, and

responsive to user selection of a hyperlink:

executing one or more functions located responsive to the step of searching; and

navigating to a display handler for display of the associated system data.

30) The method of claim 29, further comprising the step of overriding the execution of one or more functions located responsive to the step of searching.

31) The method of claim 30, wherein the step of overriding the execution of one or more functions is based on the scope of such functions.

32) The method of claim 29, further comprising the step of associating a type with each registered function.

33) The method of claim 32, further comprising the step of associating a unique system ID with each registered function.

34) The method of claim 33, further comprising the step of adding a type dependant wrapper layer to one or more registered functions.

35) The method of claim 34, wherein the step of overriding the execution of one or more functions is based on the type associated with such functions.

36) The method of claim 34, wherein the step of overriding the execution of one or more functions is based on the unique system ID associated with such functions.

37) The method of claim 29, wherein the step of altering the characteristics of text or text portion is based on a system-wide hyperlink dictionary.

38) The method of claim 29, wherein the step of altering the characteristics of text or text portion is based on a user hyperlink dictionary.

39) The method of claim 34, wherein the step of executing one or more functions includes executing a function associated with a specific system ID when present at a given scope.

40) The method of claim 29, further comprising the step of creating one or more hyperlink dictionary(s) by traversing one or more hierarchical databases.

41) The method of claim 40, further comprising the step of activating one or more hyperlink dictionary(s).

42) The method of claim 41, wherein the step of altering the characteristics of text only includes using active hyperlink dictionary(s).

43) The method of claim 42, wherein the step of activating one or more hyperlink dictionary(s) includes activating hyperlink dictionary(s) based on user input to the user interface.

44) The method of claim 29, further comprising the step of invoking other named logical actions responsive to input to the user interface.