Building Tailorable Hypermedia Systems: the embedded-interpreter approach

Kaj Grønbæk
Computer Science Department, Aarhus University
Ny Munkegade 116, DK-8000 Aarhus C. Denmark

Jawahar Malhotra
StarBase Corp., 18872 MacArthur Blvd., Irvine, CA 92715

Abstract

This paper discusses an approach for developing dynamically tailorable hypermedia systems in an object-oriented environment. The approach is aimed at making applications developed in compiled languages like Beta and C++ tailorable at run- time. The approach is based on use of: 1) a hypermedia application framework (DEVISE Hypermedia), and 2) an embeddable interpreter for the framework language. A specific hypermedia system is instantiated from the framework with the interpreter embedded in the executable. The specific hypermedia system has a number of "open points" which can be filled via the interpreter at run-time. These open points and the interpreter provide sufficient support to allow tailoring at run-time as well as compile-time. Among the types of tailoring supported are: 1) adding new media-types, 2) alternating editors for supported media-types, and 3) removing a supported media-type. The paper describes the framework and illustrates how the interpreter is integrated. It describes steps involved in tailoring a specific hypermedia system with a new drawing media-type, where graphical objects can be endpoints for links. Since the hypermedia framework uses a persistent object-store, a solution for handling persistent interpreted objects is presented. Finally, the approach is compared with other environments supporting tailoring.

OOPSLA 94- 10/94 Portland, Oregon USA

1 Introduction

Terms like 'adaptability', 'customizability', 'extensibility', and 'tailorability' are used to describe systems offering their users some potential to change the system. The use of 'tailorability' in this paper covers all of these terms. Trigg et al. [TMH87] introduced the term 'tailorability' to the area of hypermedia. They call a system tailorable if it allows users to change the system, e.g. by building accelerators, specializing its behavior, or by adding functionality. Their work illustrates changes to the behavior of the NoteCards system by using its Application Programmer's Interface (API), and the addition of new media-types by means of the NoteCards card-type mechanism. In this sense, tailoring implies adding and modifying a delivered system at the source-code level; this was possible because NoteCards was built in the Interlisp environment [MT81] where any system function is accessible, and modifiable, at any time.

In contrast, the extensible Intermedia [Mey86] system was built as a hypermedia framework specialized from the general MacApp Application Framework [Sch86]. The Intermedia framework provides support for hypermedia designers to construct their own variants of Intermedia by adding new media-types as specializations of classes from the framework. MacApp was written in Object Pascal[1], and such an extension required access to

[1]Now a C++ version is also available.

the Intermedia source code, and a compiler, in order to perform the necessary recompilation.

Both NoteCards and Intermedia support tailoring of hypermedia systems, but they both assume that the tailor has the same access to the entire development environment as the original developer had. This assumption is both undesirable and unrealistic in most use settings: the full development environment is incomprehensible, delivering source code raises legal as well as maintenance problems, a full development-environment license is expensive, etc. Thus we face some serious problems: How do we support source-code-level tailoring of systems without delivering entire development environments? How can such tailorability be provided for compiled-language environments, e.g. Beta or C++, aimed at programming-in-the-large?

This paper discusses an approach to overcome these problems and provide source-code level tailorability, at selected "open points" [Nø92], in systems built with the Mjølner Beta environment. This approach allows users to extend hypermedia systems in a fashion similar to that of NoteCards and Intermedia, with the power to do similar things, but without the need to have access to the full development environment. Although this paper elaborates the approach in the hypermedia domain, the authors believe that the approach can be applied to other application domains as well.

The DEVISE Hypermedia (DHM) framework [Grø93, GT94, GHMS94] formed the basis of these tailorability experiments. The DHM framework consists of a set of generic classes providing an object-oriented implementation of the Dexter model concepts [HS94]. The Dexter model is a general model that describes the data and functional model of hypermedia systems. The generic classes in the DHM framework provide a conceptual schema for the object structures to be stored persistently, as well as classes to handle the runtime behavior of the link-following and browsing operations. The DHM framework is built using the Mjølner Beta environment which is based on the strongly-typed, block-structured, object-oriented programming language Beta [MMPN93].

The original DHM framework supports development and tailoring of hypermedia systems by writing specializations of the generic classes; this development and tailoring requires access to the Beta compiler and the source code of the system being tailored. Hence the original DHM framework leaves one in a situation similar to NoteCards and Intermedia — there is a need for the development environment and source code of the original system. The situation is rectified by instantiating the DHM framework into a specific system which includes the Beta-interpreter library [Mal93a]. In this system, it is still possible to specialize the generic classes; this is handled by the embedded interpreter. There is no longer any need for the development environment or the complete source code of the system; only the interfaces of the classes being specialized during the tailoring process are needed. As a result, tailorability is transferred from the domain of the framework users (i.e. the developers) into the domain of the system users (most likely super-users with programming knowledge).

The paper explores a scenario in which a user of the DHM system wants to include drawings, made with an independent drawing editor, into hypermedia documents with support for links to the individual elements of a drawing. The user would, e.g like to be able to link the name of a room in the textual description of a tour of a house to the corresponding graphical object in a plan of the house. The entire process for accomplishing this is illustrated — including the code that the user has to write as well as the user's interaction with the hypermedia system to install this new code.

The paper also resolves the apparent conflict between persistence and extensibility; it shows that an object, which is an instance of an interpreted extension class, can safely be saved into a persistent store or an object-oriented database (OODB) [ABH+92]. It presents a approach whereby a system accessing such an "extended object" from the store gets extended automatically, if necessary, with the corresponding extension class. Related tailorability issues are discussed in [Mal93a, Mal94].

Although the development of the framework and the tailorable system have been done in Beta, every attempt has been made to present the results in a language-independent manner. Hence, where possible, class diagrams are used instead of Beta code. The little Beta code that appears should be self-explanatory or explained in footnotes.

2 A tailoring scenario

This section gives an intuitive feel for the types of extensions supported by the tailorable DHM system. The tailorable DHM system is extensible in that it allows for the introduction of *arbitrary* new media-types. As an example, it is shown how the supported media-types can be extended *dynamically* to allow a new media-type comprising of drawings containing objects with various geometric shapes. The new drawing media-type allows linking to/from the individual objects in a drawing.

A hypermedia system usually has built-in support for handling (creating, displaying, linking, storing) certain types of media, e.g. text, still pictures, audio, or video. The DHM system is one such system. Figure 1 shows the original non-dynamically-tailorable DHM system with a hypermedia comprising of two components: a text component and a file component.

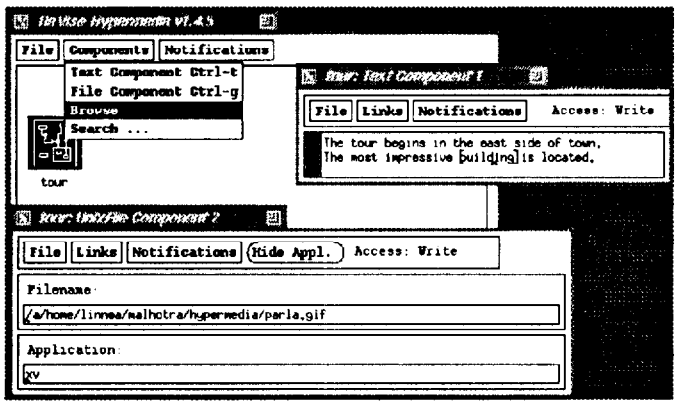

Figure 1: The original DHM system with a text component and a file component

A text component can contain arbitrary text; links can originate/terminate within the text component, i.e. substrings can serve as link end-points. A file component contains the names of a data-file and an application-file; activation of the file component (e.g. following a link to it) results in an invocation of the application-file, as a separate process, on the data-file. Links can originate/terminate at the file component, *not* within it. In Figure 1, the word `building` in the text component is a link end-point. This link terminates at the file component; following it results in the activation of the file component which results in the picture-viewing application **xv** being started as a separate process on the file `parla.gif`. A picture appears on the screen.

Thus, the DHM system has, built into it, complete support for the text media-type and limited support, via the file component, for a variety of arbitrary media-types. Arbitrary new media-types cannot be completely supported without modifying the original system and rebuilding it. To add an arbitrary new media type, you will have to get the development version of the hypermedia system, change it, and rebuild it — a task which can be both expensive and difficult.

In order to support the introduction of arbitrary new media-types, a special *dynamically*-tailorable version of the DHM system has been built. This tailorable version has, embedded within it, a Beta interpreter capable of interpreting code for new media-types within the context of the executing hypermedia system. In this tailorable version, new media-types can be defined without any need for modifying or rebuilding the hypermedia system. Given an ordinary editor for drawings, one can introduce a drawing media-type into the hypermedia system by simply writing some code and using the Extend command of the hypermedia system.

The following section illustrates, intuitively, such a tailoring scenario. It illustrates the user's interaction with the hypermedia system during the tailoring process.

2.1 The drawing media-type extension

It is easy to imagine hypermedia documents with drawings as part of them. It would be nice, for

example, to make a plan of a house, and link each room in the plan to a picture of it (as in Figure 2). Each room could also be linked to a textual description of it and vice versa.

Assume one has a drawing editor one uses to make drawings; assume also that this editor is available, either as source-code, or as an object-code library. Illustrated here is the procedure by which the tailorable DHM system is extended with this drawing editor, thus allowing one to create hypermedia documents containing drawings, with links pointing to elements within them.

Intuitively, assuming that the user has already written some additional code that 'wraps' the drawing editor, this would work as follows:

1. The user invokes Extend from the Components menu and provides the code for the drawing editor and its wrapper.

2. The hypermedia system loads the code and adds a new menu-item, called Draw, to the Components menu. The user can use this item to create drawing components.

A drawing component is illustrated in Figure 2: the snapshot shows the Components menu after the drawing component has been added to the system. A drawing component can have an arbitrary number of graphical objects, each of which has some geometric shape. These graphical objects may be end-points of links. The figure shows the scenario described earlier: a rough plan of a house is shown. There are links from the text to the individual rooms in the plan. For example, following a link from the word **kitchen** results in the appropriate room in the plan being highlighted. It is also possible to follow the link backward from the room in the plan to the corresponding word in the text component. In short, the drawing media-type is *completely integrated* with the hypermedia system.

A hypermedia document containing drawing components can also be saved in, and reloaded from, the object-oriented database, just as a normal hypermedia document without any extended

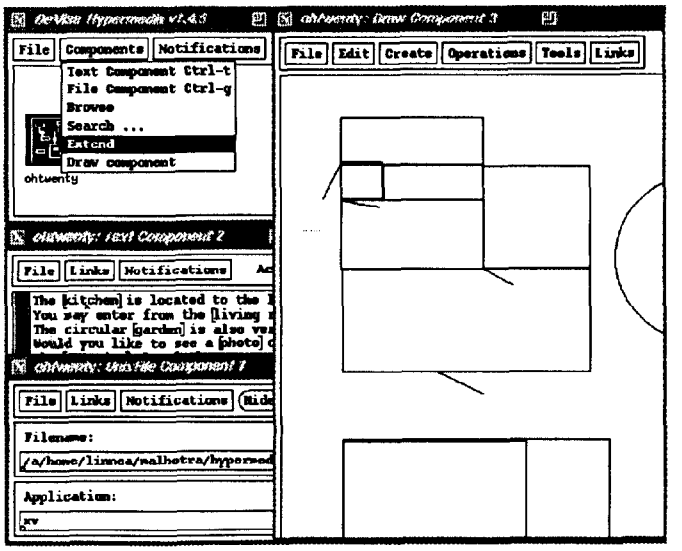

Figure 2: The DHM system integrated with a drawing editor

drawing components. Section 5 illustrates the mechanism for handling this.

As a result of this process, the hypermedia system has been extended with a new media-type for which there was no direct built-in support; all of the support was dynamically loaded. The extended hypermedia system will provide complete support for the new media-type with links to and from the internals of the media-type. Installation of the extension requires no change to the hypermedia system; in fact, the extension happens dynamically.

In this manner, the DHM system can be tailored to support arbitrary media types. In order to introduce a new media-type, the user has to: 1) write the code to support the media-type, 2) and perform the above extension-process which essentially loads the support-code into the hypermedia system.

The following sections examine the details of the implementation that contribute to making such an extensibility mechanism work.

3 DHM Tailoring architecture

The DHM system is built from a Dexter-based hypermedia framework [GT94, GHMS94]. The architecture of DHM is depicted in Figure 3. The

architecture is divided into four high-level layers: Storage Layer, Runtime Layer, Presentation Layer, and Application Layer.

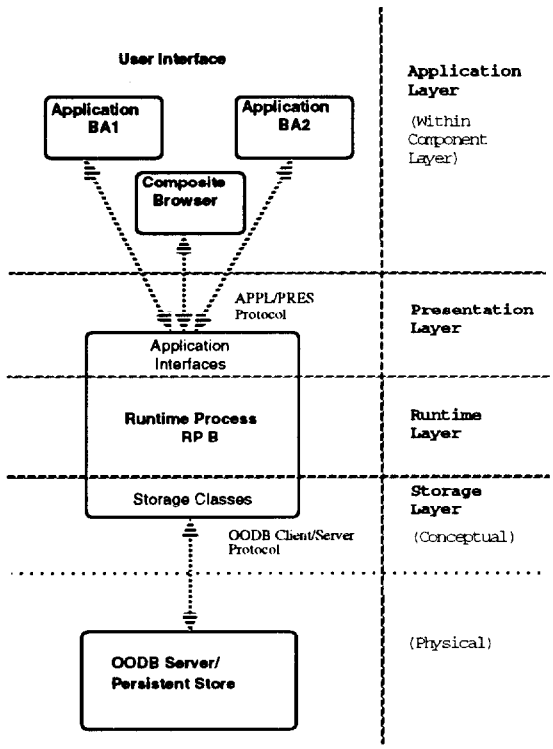

Figure 3: The architecture of DHM systems

The Storage layer corresponds to the Storage layer of the Dexter Hypertext Reference Model [HS94] and handles "permanent" data, that is, structural and content information saved in the shared database. The Runtime layer also corresponds to the Runtime layer of the Dexter model and handles information of a transient nature (used only at runtime) as well as supporting hypermedia procedures involving component-creation, link-following, etc. The Presentation layer is a layer introduced to provide an encapsulation of platform-specific code. The Application layer covers the "Within-component" layer of Dexter. The Presentation layer supports access/communication to and from (external) applications and editors belonging to the Application layer.

The Storage and Runtime layers are written completely in Beta, but are otherwise fully platform independent. The Presentation layer is also written in Beta; its interface is platform independent, while its implementation is platform and application dependent. The Storage layer may be separated into a client/server configuration (using the OODB; see Figure 3) or built into the runtime process (using the persistent-store library).

The Presentation layer defines a protocol for communication with the Application-layer process. The protocol is implemented via direct invocation for applications built into the runtime process; for external applications written in Beta, it is implemented with the Beta distribution facilities; it may use sockets on Unix, AppleEvents on the Macintosh, and DDE on MS-Windows to support communication with non-Beta external applications. Thus the Application layer may be written in any language that can support the implementation of a communication protocol with the Beta code in the Presentation layer.

One possible configuration for the system as a whole is with the Application-layer applications built into the runtime process and the Storage layer separated into a client/server configuration (OODB). In this case the APPL/PRES protocol boils down to direct method-invocation. This is the configuration used for this work.

3.1 Framework Classes

The underlying development framework consists of sets of classes to implement the three within-hypermedia layers: Storage, Runtime, and Presentation (see Table 1).[2]

It is beyond the scope of this paper to explain all the details of the framework as outlined in Table 1; however, in order to give an idea of how one can develop specific systems, or extend existing systems, a brief description is needed.

[2]Notation: Indentation indicates block-structural nesting of classes. Class names in plain text describe generic classes, whereas names in italics describe examples of specialized or application-specific code that can be added to develop a specific hypermedia system. Names in angular brackets (e.g. `<list-of-Component>`) denote the attributes of the classes they appear within.

Storage Classes	Runtime Classes	Presentation Classes	*Application Layer*
		Presentation <Appl-ref> <Protocol>	
StorageMgr	SessionMgr <StorageMgr-ref> <sessionMgrPres-ref> <list-of-Session>	SessionMgrPres <SessionMgr-ref>	*(DHM mainwindow)*
Hypertext <list-of-Component>	Session <Hypertext-ref> <sessionPres-ref> <list-of-Instantiation>	SessionPres <Session-ref>	*(Hypertext icon in DHM mainwindow)*
Component <BaseComponent-ref> <list-of-Anchor> <PresSpec-ref>	Instantiation <Component-ref> <instPres-ref> <list-of-LinkMarkers>	InstPres <Instantiation-ref>	*(Editors for component contents)*
Anchor	LinkMarker <Anchor-ref> <LinkMarkerPres-ref>	LinkMarkerPres <LinkMarker-ref>	*(e.g. brakcets around text in a TextEditor)*
BaseComponent			
LinkComponent <list-of-specifier>	LinkInstantiation <linkComponent-ref>		
Specifier <Component-ref> <Anchor-ref> <PresSpec-ref>			
CompositeComponent <list-of-Component>	CompositeInstantiation <compositeComp-ref>		*(XIconBrowser)*
	SubInst <Component-ref> <SubInstPres-ref>	SubInstPres	*(Icon in XIconBrowser*
AtomComponent *(TextComponent* *FileComponent* *etc.)*	*(TextInst* *FileInst* *etc.)*	*(TextPres* *FilePres* *etc.)*	*(TextEditor* *FileWindow* *etc.)*

Table 1: An outline of the relationship between framework classes (see footnote 2 for notation)

The *Storage layer* classes, constitute the conceptual schema for the objects stored in the Persistent Store/OODB. In the Storage layer, the **StorageMgr** class manages the storage of **Hypertext** objects. A **Hypertext** object holds a set of components. A **Component** is an abstraction that implements both "nodes" and links of traditional hypertext systems. A component holds some contents and a set of anchors, each of which identify a location within the component's contents (e.g. a part of a text).

The **LinkComponent** sub-class implements links, the **AtomComponent** sub-class implements "nodes" with simple data contents like text. **TextComponent** and **FileComponent** are examples of specific **AtomComponents** added to a hypermedia system.

In the *Runtime layer*, the **SessionMgr** class manages sessions with multiple hypertexts. The **Session** class possesses the runtime behavior for a hypertext. The **Instantiation** class which is nested within the **Session** class implements the runtime behavior of components, e.g. platform-independent user-interface control belongs here.

The *Presentation layer* has a generic **Presentation** class that acts as a superclass for all presentations; it possesses a reference to an editor or some other user-interface object of the Application layer. The **Presentation**-class attributes represent the general protocol for communication between the user-interface and the within-hypermedia objects. For example, the **instPres** class wraps an editor for a component's contents. Communication between editors and the within-hypermedia objects can be implemented via various inter-application communication facilities depending on the platform.

3.2 Adding new media-types

The purpose of the DHM framework is to support the rapid construction of hypermedia systems aimed at different application domains. The generic framework-classes provide extensive, and general, hypermedia functionality which can be reused by developing specialized classes suiting the needs of the specific domain.

The framework has been organized such that a hypermedia designer can bring in a new media-type by providing an editor for the new media-type and by building specialized classes for each of the Storage, Runtime and Presentation layers. Having built the classes the user needs to register them in the sessionMgr's type-table. This registration is done by means of typeInfo objects (see Table 2);[3] a typeInfo object is a kind of meta-object containing information about a particular class. Each of the specialized classes is encapsulated within a corresponding typeInfo object along with additional information about the class. This typeInfo object is then registered in the sessionMgr's type-table.

The typeInfo objects exist in parallel inheritance hierarchies, e.g. a textComponent's typeInfo is a specialization of an AtomComponent's typeInfo.

```
typeInfo: Class
(# attributes: attributeValueTable
   hasAttr: (# attr: Text; found: boolean;
            enter attr
            do ...
            exit found #);
   getAttr: (# attr: Text; val: Object;
            enter attr
            do ...
            exit val #);
   setAttr: (# attr: Text; val: Object;
            enter (attr,val)
            do ... #);
   removeAttr: (# attr: <text-ref>;
            enter attr
            do ... #);
   init: virtual (# do ...#);
#)
```

Table 2: Pseudo-Beta description of the typeInfo class (see footnote 3 for an explanation of syntax)

The typeInfo class contains an attribute-table consisting of (name,value) pairs; there is a general procedure interface for manipulating this table. This makes all typeInfo objects look similar to the sessionMgr; only the actual registration of attributes varies throughout the inheritance hierarchy. The attributes of a typeInfo object are typically set during initialization, but they may also be set dynamically at any time. An example of initializing the typeInfo object for the DrawComponent extension discussed in this paper is shown in Table 3.[4] In the DrawCompTypeInfo object, a name and a reference to the class DrawComponent##, among other things, are registered.

```
DrawCompTypeInfo: Class AtomCompTypeInfo
(# ...
   init: binding
   (#
     do ('Name','DrawComponent') -> setAttr;
        ('Class',DrawComponent##) -> setAttr;
        ('PrefInstTypeName','DrawInstantiation')
          -> setAttr;
        ...
   #);
#)
```

Table 3: Pseudo-beta description of the DrawCompTypeInfo class (see footnote 4 for explanation of syntax)

The Mjølner Beta environment supports classes as first-class objects in a program; hence the typeInfo objects can point to the classes they describe. This makes it possible to maintain a database of registered types, and query it for information related to the class. This table is used, for example, to compute the contents of the Components menu (see Figure 1), and to create Instantiation and Presentation objects of the right types for a given component type.

To add the drawing media-type to a DHM system, a hypermedia designer needs to provide the specialized classes shaded gray in Figure 4. Having

[3]The code is presented in pseudo-Beta syntax. Patterns are a general abstraction mechanism in Beta: they unify classes, types, procedures, and functions. `typeInfo` is a class pattern, `attributes` is a static attribute of the class, `hasAttr`, `getAttr`, `setAttr`, `removeAttr` are procedure patterns (methods) of `typeInfo`, `init` is a virtual procedure-pattern (virtual method) and hence may be further bound in specializations of `typeInfo`. In procedure patterns, the enter-part declares the formal parameters, the do-part declares the body, while the exit-part declares the return value.

[4]Here `DrawCompTypeInfo` specializes `AtomCompTypeInfo` and further binds its *init* virtual procedure-pattern (virtual method). Assignment is expressed by `->`: the value on the left is assigned to the entity on the right. Patterns with the `##` suffix denote pattern values, i.e. patterns as first-class values.

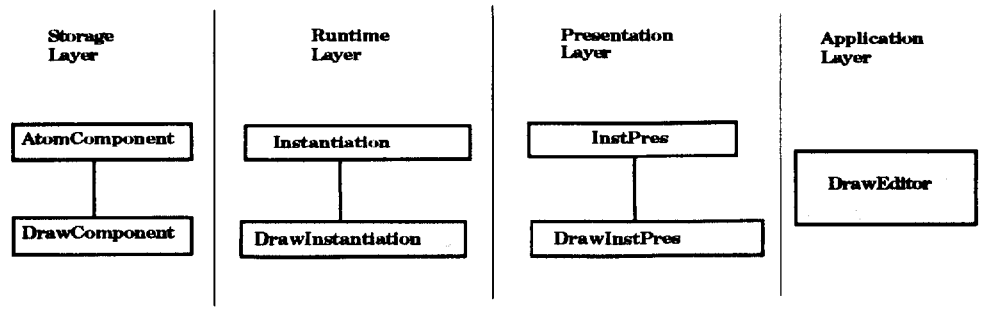

Figure 4: Classes involved in adding a Drawing media-type to the hypermedia system

developed these classes, the corresponding typeInfo objects only need to be registered with the sessionMgr's typeInfoTable. The drawing editor automatically gets included into the hypermedia system, and the drawing component becomes available to the user via the Components menu.

Observe that the typeInfo objects presented here can be viewed as kind of metaobjects [KdRB91] and their interface can be viewed as a MOP. This style of constructing a tailorable system supports Kiczales's view of building open (non-black-box) abstractions [Kic92].

3.3 Instantiating the framework into a tailorable DHM system

The DHM framework allows for the addition of new media-types by simply registering the required classes and rebuilding the system. This process requires the addition of code to register the new classes, compilation of this additional code and the new classes, and a relink of the system.

The tailorable DHM system is built from this framework by including the Beta interpreter [Mal93a]. The embedded interpreter is capable of dynamically incorporating new classes into an executing hypermedia system. The dynamically-tailorable DHM system uses the interpreter to load the classes for a new media-type and then registers them with the sessionMgr's typeInfoTable. Thus, the extensibility which was originally available at the framework level, is now available within the tailorable DHM system without any need for recompilation and relinking.

The tailorable DHM system contains the code presented in Table 4;[5] this code invokes the interpreter and registers the returned classes in the typeInfoTable.

Assuming the user has provided a name for the new component (`compName`) along with names of the files containing the Beta code for the Presentation layer (`presFile`), the Runtime layer (`instFile`), and the Storage layer (`compFile`), the above code invokes the interpreter on these files. The interpreter returns a class-pointer to the corresponding `typeInfo` class. These are instantiated and the resulting `typeInfo` objects are stored in the `sessionMgr`'s tables. These `typeInfo` objects are then initialized to set up the information in these objects. Finally, the `Components menu` is recomputed.

When the user invokes Extend (as in the tailoring scenario of Section 2), this code gets executed. The Draw menu-item then appears in the Components menu (Figure 2). When this menu-item is selected, the `sessionMgr` looks in its `typeInfo` table, finds the `DrawCompTypeInfo` object and gets all the information, like the pattern value for the `DrawComponent`

[5] The code is presented as a Beta procedure-pattern. Variables with the `##` suffix are pattern variables and can hold patterns values. The `&` means create a new instance and the `[]` suffix means take a reference to the instance.

```
extend:
(#
enter (compName, presFile, instFile, compFile)
do
  (* interpret the source code;
     get the typeInfo classes *)
  presFile -> interp -> presTypeInfo##;
  instFile -> interp -> instTypeInfo##;
  compFile -> interp -> compTypeInfo##;

  (* instantiate the typeInfo classes;
     store typeInfo objects into tables *)
  &presTypeInfo[] -> presTI[] ->
   theSessionMgr.sessionMgrTypes.append;
  &instTypeInfo[] -> instTI[] ->
   theSessionMgr.currentSession.sessionTypes.append;
  &compTypeInfo[] -> compTI[] ->
   theSessionMgr.sessionMgrTypes.append;

  (* initialize the typeInfo objects *)
  presTI.init;
  instTI.init;
  compTI.init;

  (* recompute the components menu from
     the sessionMgr's typeInfo tables *)
  theComponentsMenu.recompute;
#)
```

Table 4: Extending the hypermedia system with an interpreted media-type (see footnote 5 for explanation of syntax)

pattern, necessary to create a drawing component.

Note that variations of this are possible. For example, a variant which only changes the presentation of a given component could be written. It is also possible to support the removal of media-types: a simple way is to remove the corresponding menu-item from the components menu, but one could also imagine an open point via which the user could remove the corresponding `typeInfo` objects from the `sessionMgr`'s tables. See [Mal93b] for details on how extensions can remove functionality.

4 Code for the drawing media

The previous section showed that in order to add a new media-type to the tailorable DHM system, one has to write various classes for the new media-type and call Extend. This section illustrates the code that needs to be written; it uses the drawing media-type as an example. In Figure 4 it was shown that in order to add the drawing media-type to the hypermedia system, it is necessary to define the classes DrawComponent, DrawInstantiation, DrawInstPres, and DrawEditor. This section focuses on the DrawInstPres and DrawEditor classes as these comprise the interface between the drawing editor and the hypermedia system, and are sufficient to illustrate the idea.

The construction begins with a drawing editor. For this experiment, an existing drawing editor was used. This editor supports the creation of graphical objects with various shapes. These graphical objects have identities. There is the notion of a current graphical object; the editor can indicate the current object visually by drawing it in a different color from the other objects. The entire drawing editor is written in an object-oriented style in Beta. The relevant details of the code are examined in the following sections.

Figure 5 summarizes the construction process. The existing drawing editor (DrawEditor) is specialized into a presentation-compatible drawing editor (DrawEditorForPres). DrawInstPres, the presentation class for the drawing media-type, is defined as a specialization of InstPres.

When a drawing component is created by the user, an instance of the drawing-presentation is created. This, in turn, creates an instance of the presentation-compatible drawing editor. The DHM object invokes operations on the drawing-presentation object and vice versa; the drawing-presentation object invokes operations on the presentation-compatible drawing-editor object and vice versa.

For each drawing component in the hypermedia system, there will be a DrawInstPres and a DrawEditorForPres instance, as well as a DrawInstantiation and a DrawComponent instance which are not discussed here.

4.1 The details

As shown in Figure 5, the construction has two parallel threads: the existing editor is specialized

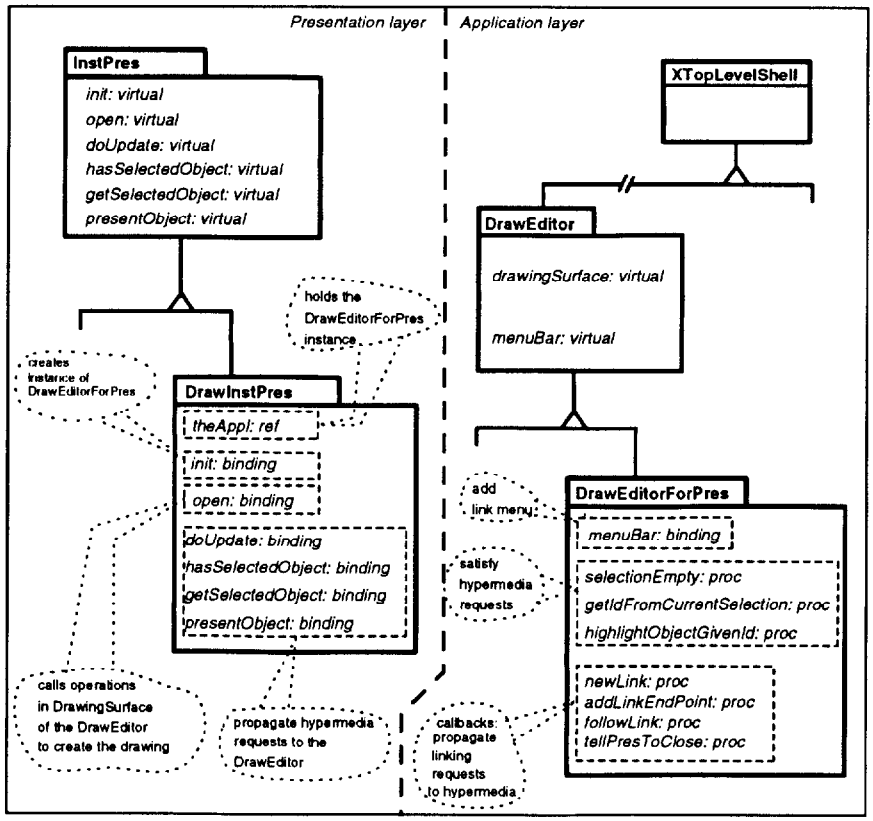

Figure 5: The support code for the drawing media-type

to make it compatible with the hypermedia system and the presentation class in the hypermedia system is specialized to make it compatible with the extended editor.

Figure 5 shows the Beta class DrawEditor; this is the original drawing editor. Nested within it are two virtual-classes: drawingSurface and menuBar. The drawingSurface implements the window in which the drawings are made. Nested within it, and not shown in the diagram, are various classes and methods which describe the different graphical shapes and their behaviors. menuBar implements the menu bar: all the menus and the items that comprise them are described within menuBar.

DrawEditor is specialized into DrawEditorForPres. In the specialization of the editor, a new menu called Link is added. This is to enable linking to and from the various graphical objects in the drawing editor (see Figure 2). It has items such as New Link Source, Add Link End-point, and Follow Link. The Quit menu-item in the File menu is replaced by a slightly different Quit Window. This is because the editor is no longer a stand-alone application; it cannot just quit as it did before; instead it must inform the hypermedia system that it is about to close and must save its contents within the hypermedia document.

These changes to the menus of the original drawing editor are accomplished by extending the menuBar virtual-class.[6] The new menu commands are implemented in terms of calls to the methods newLink, addLinkEndPoint, and followLink[7] which are defined in DrawEditorForPres.

These methods implement their behaviors by calling operations declared within the hyperme-

[6] In Beta, virtual sub-classes refine, not replace, their super-classes; they are referred to as further bindings; hence the qualification *binding* in the diagram.

[7] These are Beta patterns used as procedures (methods); hence, the qualification *proc* in the diagram.

dia system. For example, newLink calls thisSession.newLink where thisSession is an attribute declared within the class InstPres (actually a superclass of it). The name thisSession is visible within DrawEditorForPres because DrawEditorForPres is placed lexically within DrawInstPres,[8] thus making it in the lexical scope of declarations made within DrawInstPres and all its super-classes. This is an example of the power of the approach: the extension can reach into the original system and access any state-information and operations visible in its scope.

The Quit Window menu-item of the extended editor uses the tellPresToClose method, declared within the extended editor, which is implemented by a call to myInst.tellPresToClose. Here, myInst is once again a variable declared in a lexically-enclosing block. The tellPresToClose method in myInst calls doUpdate in DrawInstPres. This scans all the graphical objects in the current drawingSurface and writes them into the OODB as the contents of this drawing component.

The methods selectionEmpty, getIdFromCurrentSelection, highlightObjectGivenId are also defined within DrawEditorForPres. These are implemented in terms of simple operations within the drawingSurface; they get called by the presentation object.

Figure 5 also shows the DrawInstPres class as a specialization of InstPres, which is built into the hypermedia system. This is an example of an extension which specializes a built-in class. In fact, in the tailorable DHM system the presentation object is partly compiled and partly interpreted: the behavior inherited from InstPres is compiled while that defined in DrawInstPres is interpreted.

DrawInstPres doesn't add new operations to the interface of InstPres; it merely defines the implementation of the operations already defined in InstPres. The init method is defined to create an instance of the DrawEditorForPres and to store that instance in the reference-variable theAppl.

The open operation gets invoked by the hypermedia system when an existing drawing component is to be displayed. Its purpose is to make the editor display the stored drawing. This drawing is stored in the OODB (by doUpdate). The open operation accesses the drawing elements from the OODB and calls operations in the drawingSurface of the current drawing editor to display them.

The doUpdate, hasSelectedObject, getSelectedObject, and presentObject operations form the general protocol for the hypermedia system to get information from the drawing editor.[9] They are implemented in terms of operations selectionEmpty, getIdFromCurrentSelection, and HighlightObjectGivenId defined within the editor. These definitions are sufficient to interface the editor to the hypermedia system.

4.2 Typical execution sequences

Say the user creates an object in the drawing editor. The creation events are routed to the drawing editor's event handler via the window-system toolkit. Hence, the drawing editor behaves just as it did outside the hypermedia system.

Now, the user selects NewLink[10] from the Links menu of the drawing editor. This is handled first by the drawing editor; it calls the newLink method defined within it, which calls the newLink operation in the current-session object in the hypermedia system. This queries the drawing editor, via the corresponding presentation object, for the currently-selected object in the editor. Given this object (or its ID), it creates a link originating within this drawing component at the selected object.

The user then follows a link to an inactive drawing component (e.g. from a text component). The link contains a reference to the contents of the drawing component and to a single object within it. The hypermedia system first creates an instance of the drawing presentation (DrawInstPres) and initializes it; this results in a drawing editor. It then accesses the contents of the drawing component from the OODB, calling open on the presentation object. This results in the display of the drawing component contents in the drawing editor. It then

[8] the lexical placement is not shown in the diagram.

[9] and, in general from any editor.

[10] Assume the user is creating the source of a new link.

asks the presentation object, via the presentObject method, to display the object at the link end-point.

This section has illustrated the type of coding effort required in order to extend the hypermedia system with a new media-type. Although shown for the drawing media-type, the techniques apply equally well for other media-types.

5 Extensions and Persistence

The DHM system uses the persistent store (or the OODB) to store hypermedia documents. When the system is extended with a new media-type, like the drawing media-type, interpreted drawing components will be part of the saved hypermedia documents. Hence, it must be possible to save these interpreted objects into the store as well as reload them from the store.

When an object is saved into the persistent store (or the OODB), a reference to its prototype (behavior) is saved along with its state. A prototype, in Beta terminology, is a runtime descriptor of the class. When the object is loaded into an executing program reference is used to locate the prototype of the object in that execution. Normally, a program saving the object has its prototype compiled into it, and a loading program is also expected to have the prototype compiled into it.

What happens, then, in the case of an object whose prototype is interpreted, or in general, dynamically loaded? When saving the object, the prototype will generally be present in the execution. But, when loading it, its prototype may or may not be available. If the extension has already been installed, then its prototype will be there, else it won't. So, the question is: what happens when an object is loaded and its prototype is not present in the execution?

A satisfactory answer is: load the prototype dynamically, automatically and transparently. Shown here is an approach for making this work. The approach assumes that the extensions are interpreted, but it should work, with slight modifications, even if they are dynamically linked.

In order to explain the special approach, it is necessary to introduce some of the basic functionality of the persistent store. When an object is saved, a reference to its prototype is saved by saving the name of the object file (module) in which it was defined, along with an index which uniquely identifies the prototype within that file. When the object is accessed, possibly in another program, the name of the object file and the index are used to locate the prototype in the accessing program's current execution's address-space. The name is used to locate the object file in the accessing program, and the index to locate the prototype within that file.

This simple scheme is implemented by ensuring that every program using the store has a table describing all its object files. This table is built when the store is initialized. Intuitively, this table has one entry for each object file in the program it resides within. Each entry describes the corresponding object file; it allows a prototype to be mapped into a unique index and vice versa.

Figure 6(A) shows this table, its entries, and the relevant classes; only relevant details are shown. The table is an instance of the class ExecGroupTable (EGT). The entries are instances of CompExecGroupTableElement (CompEGTElement). The table has a method findByName, used to locate an entry by the name of the object file. The first step in saving or accessing an object is to determine which file it belongs to, and then call findByName on the table to access the corresponding entry. Once the entry has been obtained, its methods protoToIndex and indexToProto can be used to get the unique index for the object's prototype (saving), or to get the prototype corresponding to the unique index (restoring).

5.1 Handling persistence for interpreted classes

A program that wishes to save interpreted objects into the persistent store must use a specialized version of the table. This is illustrated in Figure 6(B). The specialized table is an instance of InterpEGT, which is a sub-class of EGT. In this specialized version, the findByName method is extended to handle

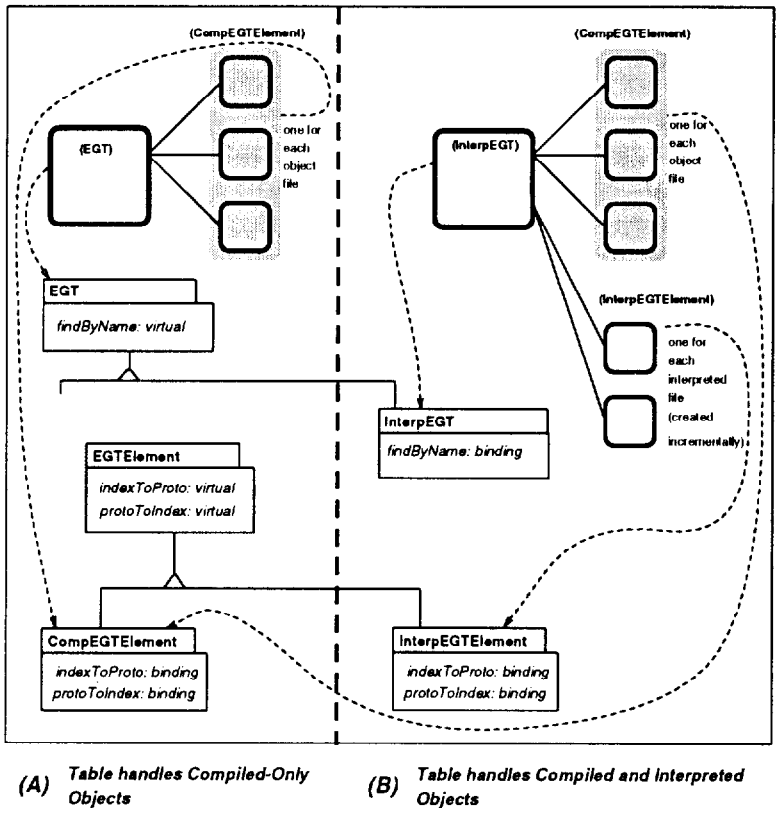

Figure 6: The tables for the persistent store

unsuccessful searches, i.e. searches for files that its super-class cannot handle. Its super-class will be able to handle all searches for files that are compiled, leaving only the interpreted ones to it.

This specialized **findByName** is designed to invoke the interpreter on the file being searched for. This process creates prototypes, in the current address-space, for all the classes declared in that file. **findByName** then creates a specialized entry representing that file and returns it. The specialized entry is an instance of InterpEGTElement, which is a sub-class of EGTElement. In order to record the fact that this file has been interpreted, and to be able to service future requests for the same file, **findByName** also saves this entry in the table, separated from the compiled entries. This is illustrated in Figure 6(B), where the table has two sets of entries.

The **protoToIndex** method of this specialized entry behaves just as it does in normal entries (i.e. instances of CompEGTElement). The **indexToProto** method determines where the prototypes for this interpreted file reside in the current address-space, it maps index to address and returns it.

Saving an interpreted object for the first time results in creation of an element for its file. Accessing the object from the store triggers the extended **findByName** to interpret the corresponding file if not already done. Note that an object saved as interpreted can be loaded with compiled prototype in an executable where its file has been compiled.

Observe that in order to make this approach work, the only parts of the original persistent store that have been specialized are the EGT and EGTElement.

5.2 Loading persistent interpreted components

Returning to the DHM system, when a hypermedia document containing a drawing component

is saved, only the *component* object (instance of DrawComponent) is saved in the store. The presentation and instantiation objects are not saved.

When this hypermedia document is later accessed, loading the drawing component object triggers a process by which, as described above, the drawing-component class (DrawComponent) — and its typeInfo class, which is defined along with it — will get interpreted and transparently installed into the system. The instantiation and presentation classes, DrawInstantiation and DrawInstPres, will, however, not get defined via this mechanism. These are necessary in order to handle the drawing-component object at run-time. The loading of the code for these classes should also be transparent to the user of the hypermedia system.

It is accomplished by extending DrawCompTypeInfo (shown in Table 3) to have attributes which specify the locations of the source-code files for the instantiation and presentation classes. The SessionMgr uses these attributes to trigger the interpretation of the source code files, resulting in the definition of the DrawInst and DrawInstPres classes.

6 Extensions and new versions

One of the problems with using extensible systems is that extensions made on one version of the system are often incompatible with a newer version of the system [Mac91]. Extensions written for tailorable systems built using this approach are much more likely to survive upgrades to new versions of the system. This is primarily because the extensions do not have unrestricted access to the system. Hence, the new version of the system has only to ensure that it still has the correct open points. Furthermore, should a new version of the system have changes to the original open points — say, changes to the interface of an open point, or deletion of an original open point — the type system will flag the incompatible extensions when they are loaded into the new version. It will also most likely help the user identify what changes should be made.

The following example illustrates another reason why extensions in this approach are more likely to survive new versions of the tailorable system. Assume that a system has a class P which has a virtual method A within it. Now assume that the user defines a sub-class Q of P. In Beta, the user may only further-bind the method A, i.e. extend its functionality; the user cannot replace the original A. When the user does this, the user *need not* make any assumptions about all the places where A may be used. The only assumption the user makes is about the *extension-interface*[11] of A in order to determine how it can be extended. On the other hand, if the user were able to replace A, the user *would have* to make assumptions about all the places where the original A may be used.

Suppose, a new version of the system is released, and in this new version, the A is used in a new way. This new use couldn't possibly have been in the set of assumptions made by the replacement user as it didn't exist when the user wrote his/her extension. As a result, installing the old extension into the new version will most likely have unexpected results. Not only this, it will probably be very difficult to find the exact cause of the problem. On the other hand, the further-binding user will be much better off; as long as the extension-interface of A remains the same in the new version, the old extension will work in the new version.

7 Concluding remarks

An approach for building dynamically-tailorable systems in a compiled language environment has been shown. This approach is based on having a generic framework for building systems in the relevant domain, and instantiating it into a specific system that includes an interpreter for the framework language. In addition, the specific system has "open points" where new classes can be installed. The resulting system is then tailorable in a manner similar to systems which run in residential environments like Lisp and Smalltalk. It supports extensions to the underlying framework at runtime.

[11] By extension-interface is meant only those aspects of a class signature and behavior affecting extensions of it.

The approach has been demonstrated for hypermedia systems. The construction of a dynamically tailorable hypermedia system has been shown. The tailoring process in order to install a new drawing media-type has also been shown. Problems encountered in saving interpreted objects into a persistent store or OODB have been resolved.

It is important to note that in this approach, the designer of the system has control over where the open points should be, and as a result, has control over what should be tailorable. Thus, the resulting tailorable systems do not allow users to make arbitrary changes to the system, as can be done in Lisp and Smalltalk environments.

An advantage, which is also a limitation, of open-points in Beta is that extensions written by the user cannot easily break functionality already present in the system. This is mainly due to the semantics of Beta, and is discussed in great detail in [Mal94]. In Beta, sub-classes are true extensions of their super-classes; a sub-class refines its super-class, it does not replace parts of its super-class. This limits what can be done by the extension, thus limiting the damage-potential of the extension.

A tailorable system built using this approach will probably *not* be tailorable by a non-programmer user. As has been demonstrated in the paper, the tailoring process requires a small coding effort and a reasonable command of programming. Most use-sites, however, have super-users who could possibly be able to do this kind of tailoring [Mac90]. An extension to this approach would be to build a specialized application-oriented language and environment atop these open points; this would reduce the coding effort. So, the tailorable system, instead of asking the user for Beta source-code files could provide an environment for the user to construct the code. This environment could be specialized to, say, the hypermedia domain, and provide constructs to directly support hypermedia-system extensions, e.g. the definition of new media-types.

To tailor the DHM system with the new drawing media-type, the user must have available, in addition to the system, the interfaces of the framework classes used. In addition, symbol-table information, generated during the compilation process of the original system, must be available for the interpreter to locate the object-code for the framework classes. This is all the information about the original system that is necessary. In our experiments, we had available, annotated abstract syntax-trees of the original system. These had more information in them than was really necessary. It should be possible to construct trimmed versions of these trees with only the necessary information. In fact, a prototype implementation of this is available as part of the Mjølner Beta system. So a tailorable system, built using the approach, must be shipped with this additional interface and symbol information in order to be tailorable. To compare the overhead of the embedded-interpreter approach with other approaches for providing tailorable hypermedia systems, we list measures for different tailoring approcahes in Table 5.

The embedded-interpreter overhead adds up to 3.8 MB. The size increase caused by embedding the interpreter (1.3 MB) can definitely be further reduced. The value presented is for the first unoptimized version of the interpreter. In addition, forthcoming improvements in the Beta compiler show promise of producing smaller object-code files.

The Beta compiler produces three symbols for each pattern it compiles, while the interpreter uses only one of these. Hence, the number of symbols can be reduced by a third; thus, the symbol-table should occupy only 0.8 MB. This compiler improvement is also forthcoming. Thus, the revised overhead is only 2.3 MB.

This overhead of less than 2.3 MB appears small in comparison with that of other source-code-level-tailorable systems. For instance, the hypermedia systems Intermedia and NoteCards discussed in this paper seem to require a much larger overhead to provide similar tailorability. Intermedia was built using MacApp and MPW; major parts of these basic environments are needed, along with the Intermedia code, in order to tailor Intermedia. These core parts of MacApp and MPW in themselves, by rough measurements, add 15 MB in overhead in terms of libraries and executables necessary

Interpreter Approach			Compiler Approach		
	untrimmed symbol table	trimmed symbol table		static link	incremental link
Interpreter	1.3	1.3	Compiler	3	3
Symbol table	2.3	0.8			0.8
Interface files	0.2	0.2		0.2	0.2
			Assembler	0.15	0.15
			Linker	0.1	0.1+
			DHM object code	3	
			Libraries	1.2	
TOTAL	3.8	2.3		7.65	4.25+

Table 5: Interpreter vs Compiler overhead comparisons (sizes in MB).

for tailoring a system like Intermedia. NoteCards was built in the residential Interlisp environment; thus the entire environment is necessary both to run and to tailor NoteCards. An Interlisp image typically adds 12 MB in overhead for a system like NoteCards to be executed and tailored.

The overhead of 2.3 MB can also be compared with that of tailoring the DHM system using the full Beta development environment. In this scenario, the user would create the extensions as shown here, compile them, and relink the application. This give a total overhead of 7.65 MB, or 333% of the interpreter-approach overhead.

Yet another scenario can be considered. Here, the compiler would be used to compile the extension, and an incremental linker used to link the extension into the DHM executable. In such a scenario, the overhead would be 4.25 MB, or 184% of the interpreter approach overhead.

Even though the drawing editor, and its presentation class, are interpreted, their performance is acceptable. It, however, remains to be seen how the interpreter would perform in the case of a cpu-intensive media-type like video. The time taken to extend the hypermedia system — with the built-in interpreter — with a new media-type is, however, a small fraction of the time it would take to compile the necessary sources and relink the system. In [Mal94] the excution of interpreted code is compared with that of compiled code.

It has thus been shown that we can provide close to the level of tailorability of systems like NoteCards and Intermedia, but at a fraction of the overhead, and with potentially greater safety.

Acknowledgements

This work has been supported by the Danish Research Programme for Informatics, grant number 5.26.18.19, and the ESPRIT II/III projects EuroCoOp and EuroCODE. We greatly thank Randy Trigg for his contributions to the development of the DHM framework. Thanks to Lennert Sloth for helping with problems in getting the tailorable hypermedia system working. Thanks to Søren Brandt for help with the persistent store.

References

[ABH+92] P. Andersen, S. Brandt, J. A. Hem, O. L. Madsen, K. J. Møller, and L. Sloth. Distributed Object-Oriented Database Interface. Technical Report No. ECO-JT-92-3, Jutland Telephone and Aarhus University, 1992.

[GHMS94] K. Grønbæk, J. A. Hem, O. L. Madsen, and L. Sloth. Designing Dexter-based Cooperative Hypermedia Sys-

tems. *Communications of the ACM*, 37(2), February 1994.

[Grø93] K. Grønbæk. Object oriented model for the Distributed Hypermedia Toolkit. Technical Report No. CODE-AU-93-10, Aarhus University, Computer Science Department, 1993.

[GT94] K. Grønbæk and R. Trigg. Design issues for a Dexter-based hypermedia system. *Communications of the ACM*, 37(2), February 1994.

[HS94] F. Halasz and M. Schwartz. The Dexter Hypertext Reference Model. *Communications of the ACM*, 37(2), February 1994.

[KdRB91] G. Kiczales, J. des Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, 1991.

[Kic92] G. Kiczales. Towards a New Model of Abstraction in the Engineering of Software. In *Proceedings of IMSA'92 (Workshop on Reflection and Meta-level Architectures)*, Tokyo, Japan, November 1992.

[Mac90] W. Mackay. Patterns of Sharing Customizable Software. In *Proceedings of CSCW '90*, Los Angeles, CA, October 1990. ACM Press.

[Mac91] W. E. Mackay. Triggers and Barriers to Customizing Software. In *Proceedings of CHI '91*, pages 153–160. ACM Press and Addison Wesley, April 1991.

[Mal93a] J. Malhotra. Dynamic Extensibility in a Statically-compiled Object-oriented Language. In S. Nishio and A. Yonezawa, editors, *Proceedings of the International Symposium on Object Technologies for Advanced Software (ISOTAS'93)*, LNCS 742, pages 297–314, Kanazawa, Japan, November 1993. Springer-Verlag.

[Mal93b] J. Malhotra. Extensibility as the basis for Incremental Application Generation. Technical report, Computer Science Dept., Aarhus University, November 1993.

[Mal94] J. Malhotra. On the Construction of Extensible Systems. In *Proceedings of TOOLS EUROPE '94*, Versailles, France, March 1994.

[Mey86] N. Meyrowitz. Intermedia: The Architecture and Construction of an Object-Oriented Hypermedia System and Applications Framework. In *Proceedings of OOPSLA '86, Object-Oriented Programming Systems, Languages, and Applications*, pages 186–201, November 1986. Published as ACM SIGPLAN Notices, volume 21, number 11.

[MMPN93] O. L. Madsen, B. Møller-Pedersen, and K. Nygaard. *Object-Oriented Programming in the BETA Programming Language.* Addison-Wesley Publishing Company, 1993. ISBN 0-201-62430-3.

[MT81] L. Masinter and W. Teitelman. The Interlisp Programming Environment. *Computer*, 14(4):25–34, April 1981.

[Nø92] K. Nørmark. From Hooks to Open Points and back again. Technical report, Aalborg University, March 1992.

[Sch86] K. J. Schmucker. Introduction to MacAPP. In *Object-Oriented Programming for the MacIntosh*, chapter 4, pages 83–129. Hayden Book Company, Hasbrouck Heights, New Jersey, 1986.

[TMH87] R. H. Trigg, T. P. Moran, and F. G. Halasz. Adaptability and Tailorability in Notecards. In *INTERACT 87, Stuttgart, Germany*, September 1987.