AppleBuilder - an Object-Oriented Application Generator Supporting Rapid Prototyping*

Kaj Grønbæk Anette Hviid
Randall H. Trigg†
Computer Science Department
Aarhus University, Denmark

October 1991

Abstract

This paper describes an object-oriented application generator, AppleBuilder, currently being developed in the Mjølner BETA programming environment. AppleBuilder supports several rapid prototyping styles as well as final development of BETA applications. User interface objects such as dialogs, menus, and windows are designed using direct manipulation graphical editors. Actions behind buttons and menu items are programmed as "scripts" in textual editors activated from within a graphical editor. The editors reflect changes in the code directly in an underlying Abstract Syntax Tree (AST) thus saving compilation time. Moreover, generated applications are modularized so that editing, for instance the script for a button, only requires recompilation of the script itself. An advantage of AppleBuilder compared to other user-interface design tools such as HyperCard is that AppleBuilder's scripts are embedded in a general purpose programming

*To appear in proceedings of the *Fourth international conference on software engineering and its applications*, Toulouse, December 9-13 1991.

†Email: kgronbak@daimi.aau.dk, hviidand@daimi.aau.dk, and rtrigg@daimi.aau.dk.

language making it possible to avoid calls to external routines written in another language. In addition, ApplBuilder's ability to work with ASTs instead of textual code skeletons supports reverse engineering.

Keywords: Rapid Prototyping, User Interface Design, Application Generation, Object-Orientation, Graphical Editing, Meta-Programming, Reverse Engineering.

1 Introduction.

In recent years, prototyping has come to the fore as an essential part of the system development process (Squires et al, 1982; Boar, 1984; Lantz, 1986; Wilson & Rosenberg, 1988). On the one hand prototyping involves the construction and modification of partial but operational computer artifacts, on the other hand the prototypes are used as catalysts in design discussions with users. Building prototypes requires a computing environment that can support rapid incremental software development. Furthermore, as argued by Bødker and Grønbæk (1989) and Grønbæk (1991), it is advantageous to allow certain modifications to the prototype to be undertaken in the presence of users. This sort of "cooperative prototyping" extends the demands made on the computing environment (Grønbæk, 1990). These can be broadly grouped as follows:

1. *A direct manipulation graphical interface* is imperative for quickly laying out complex application interfaces. The graphical editors used to build the application's *user interface objects* (UIOs) must be able to automatically generate *code templates* so that these objects may be constructed using graphical editing only. The direct manipulation graphical interface also makes it possible to avoid much of the on-the-fly textual programming characterizing cooperative prototyping sessions (Bødker and Grønbæk, 1989).

2. *An object-oriented programming language* is crucial to facilitate stuctured programming and code sharing through inheritance. In this way, new prototypes can be created by assembling and specializing class definitions from existing prototypes and libraries of UIO implementations. The ability to represent familiar objects directly in a program's user

interface supports the users' understanding of prototypes and systems. Thus object-oriented design and programming support the cooperative prototyping style mentioned above.

3. *Support for fine-grained modularization* of code and separate compilation is essential in order that the work of modification be localized at appropriate points in the prototype (for example, in the form of *scripts* behind UIOs) and that minimal recompilation overhead be incurred. Again, this is especially important if modifications are being undertaken in the presence of users.

Though prototyping environments exist that meet some of these demands, we know of none that meets them all. HyperCard[1], perhaps the most well-known prototyping environment on the Macintosh (Goodman, 1987), owes much of its success to a highly developed graphical interface and a customized scripting language, HyperTalk. HyperTalk can be used to implement a wide range of actions "behind" UIOs like windows, fields and buttons However, HyperCard developers are all too familiar with the "wall" that must be confronted when the limits of HyperTalk are reached. When HyperTalk becomes insufficient the developers write routines in Pascal or C and call these through HyperTalk's XCMD interface (Shafer, 1988).

Another example is Prototyper™, a Macintosh tool used to generate *code skeletons* for simple user interfaces (consisting of menus and windows).[2] Unlike HyperCard, the resulting code is in a common high-level programming language (Pascal or C). A desirable feature of Prototyper™ is its ability to link UIOs together and preview the resulting interfaces. For example, a menu entry can be linked to a dialog so that the dialog is displayed when the entry is selected. However, Prototyper™ is a one-way street. Once a code template has been modified, Prototyper™'s graphical editors can no longer be used to modify it. Furthermore, it lacks the ability to manage scripts behind specific UIOs.

Other products for the Macintosh like MacApp[3] and Allegro Common-Lisp[4] are more properly considered programming environments although

[1]Copyright Apple Computer, Inc.

[2]Prototyper™ is written by George R. Cossey and it is a product from SmethersBarnes. All trade marks are acknowledged.

[3]A product from Apple Computer. All trade marks are acknuwledged.

[4]A product from Apple Computer. All trade marks are acknowledged.

their functionality overlaps with that outlined above. In order to qualify as prototyping environments, they would need to support the generation of significant portions of the target application through *graphical editing* and *automatic code generation*. User interface code written directly would need to be structured in the form of scripts and locally placed "behind" the relevant UIOs. The object-oriented visual programming system ProGraph[5] should also be mentioned. It provides graphical editing for both UIOs and application code; all programming is done by placing and connecting icons in windows on the screen. UIOs are designed in graphical editors and the actions behind the UIOs are specified as calls to methods on objects specified in the iconic code. For example, a button in a dialog has a 'Click method' field where a name of a method is entered by the ProGraph user when designing the dialog. The iconic program can be inspected and even edited interactively during program execution. However, standard textual programming is not supported by ProGraph. We consider this to be a disadvantage for experienced programmers who are generally intimately familiar with the syntax of their favourite programming languages.

The Interface Builder on the NeXT computer (NeXT) is yet another environment that meets some of the demands. It is a graphical environment for building interfaces incorporating graphics and sounds. Interface Builder generates code (Objective-C) in textual form. In addition, it supports linking graphical objects on the screen to methods implemented in existing code modules. This linking is accomplished by means of fully graphical browsers. As long as the programmer only graphically manipulates instances of existing classes, Interface Builder manages all the UIO code and keeps it separate from the rest of the application code. For classes, Interface Builder creates separate interface and implementation modules. Interface Builder manages the interface module transparently, whereas the programmer manually fills in class methods in the skeleton implementation module using a textual editor. Furthermore, the programmer is responsible for manually updating the implementation module when changes are made from the graphical editor.

The work described in this paper is aimed at meeting the outlined requirements and overcoming most of the disadvantages of existing environments in a single environment, ApplBuilder. Built on the object-oriented programming language BETA (Kristensen et al., 1987 & 1990) and making use of

[5]A product from Gunakara Sun Systems Limited. All trade marks are acknowledged.

the Mjølner BETA System's Meta Programming and Fragment Systems, ApplBuilder supports fast creation of user interface-intensive prototypes and the incremental expansion of these into complete applications. The current implementation of ApplBuilder is for the Apple Macintosh only but the principles are applicable for any kind of user-interface systems like X-windows etc. Code generation using the Meta Programming System was proposed and explored in an earlier project by Bjerregaard and Hviid (1990). Those ideas have been further developed in ApplBuilder.

In what follows, we describe the ApplBuilder environment and the various kinds of application editing it supports. We then describe the structure of applications built with ApplBuilder and the ways it may be used in rapid prototyping. We conclude with a discussion of outstanding problems and the status of the development.

2 The ApplBuilder Environments

The environment of ApplBuilder is built on the object-oriented Mjølner BETA System as shown in Figure 1. The lines connecting nodes in the figure denote use relations. In what follows, we describe each of the major components of the system.

The Mjølner BETA System.

The The Mjøher BETA System is designed to support the development of large industrial systems. It is a highly integrated grammar-based programming environment supporting object-oriented programming (Bak et al., 1991). The system is based on a powerful notion of separate compilation that enables full consistency to be ensured across compilation units; when the program is changed, only the affected compilation units are re-compiled. All modularization in ApplBuilder is done using the Fragment System (Mjølner, 1990), one of the grammar-based tools provided by the Mjølner BETA System. Fragments are the units of manipulation throughout the system, and the BETA compiler translates BETA fragments into native code. The set of fragments stored on a file are called a fragment group.

The intergation of ApplBuilder's editors is accomplished using the Mjølner BETA System's uniform representation of programs; abstract syn-

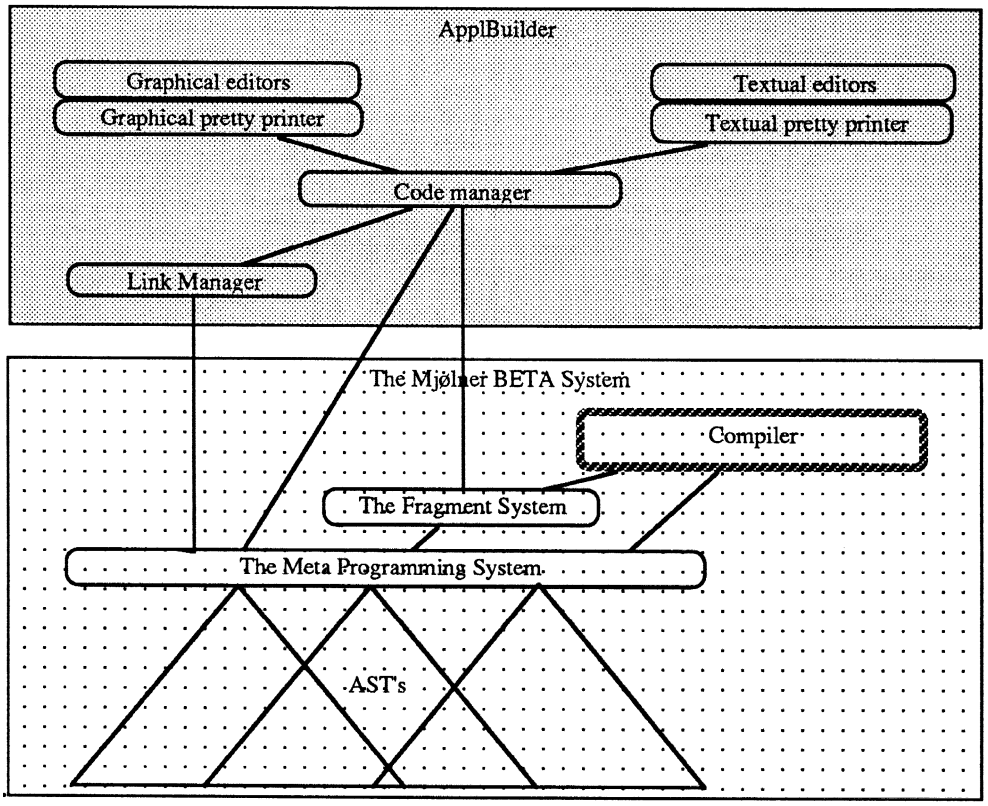

Figure 1: The architecture of the ApplBuilder environment.

tax trees (ASTs). Manipulations of the ASTs are done through the Meta Programming System (MPS) (Madsen and Nørgaard, 1987). Furthermore, all editors working on the same AST are informed when the AST changes, thereby allowing the editors to ensure consistency. We use an extension to the MPS which makes it possible to decorate the nodes in an AST with *properties*. These properties are used to store comments, link information, and layout information on UIOs to be used by the graphical editors.

The programming language being used both for implementing ApplBuilder and for the code generated by the code manager is BETA. BETA is a block structured, strongly typed and object-oriented language. It is intended for describing program executions regarded as models of objects and concepts. The expressiveness of the BETA language is achieved by focusing

on simplicity, abstraction and orthogonality.[6]

Editors.

There are two kinds of editors in ApplBuilder: graphical editors and textual editors. When a piece of the application code is to be edited it is transformed ("pretty-printed") from a uniform representation, the AST, into either a graphical or a textual representation. The AST is represented as text by means of a conventional pretty-printer which uses a grammar-based specification to guide the format of the output. Graphical pretty-printing is done by interpreting special properties attached to the AST. The code manager determines whether the code should be pretty-printed graphically or textually depending on the user's editing request. That is, on what level of abstraction editing is to be done and what kind of code is to be edited. A user interface object, for instance, may be edited both graphically and textually.

The Code Manager.

Code generation and switching between editors are handled by a code manager. The code manager generates several kinds of code: (1) skeletons for new applications when starting from scratch, (2) skeletons for UIOs each time a new one is requested and (3) pieces of code each time the graphical layout of a UIO is changed.

The Link Manager.

Following Sandvad (1989), ApplBuilder includes a link manager that maintains connections between fragments for the fragment browser and links between the declarations of UIOs and their instantiations. Thus, a designer viewing a UIO instantiation in a textual editor can immediately bring up

[6]There is only one single abstraction mechanism in BETA called a pattern. A pattern is a generalisation of the common constructs: class, type, procedure, function, and method. The result is a uniform treatment of all these concepts. BETA also contains virtual patterns which are generalisations of virtual procedures as seen in Simula and C++. Several other linguistic notations such as nesting and block structure are streamlined. For further information on BETA, see Kristensen et al. (1987 & 1990).

its declaration viewed with a graphical editor. In the future we expect to develop the link manager to provide more general hypertext facilities similar to those down from powerful hypertext systems like NoteCards (Halasz et al, 1987) etc.

3 Application Editing

To fulfil our vision on application generators, ApplBuilder provides several types of support for editing programs and automatic code generation. In the following we consider three ways of editing application code supported by a variety of integrated editors: configuration editing, graphical editing and textual editing. Use of the editors is described using figures from a calculator application developed in ApplBuilder.

3.1 Configuration Editing

An application is organized as a collection of fragments which can be browsed and edited using a fragment browser. As long as all configuration editing is done within the fragment browser, ApplBuilder can manage the organization of the application for the designer. For new applications, a minimal configuration of fragments is created automatically. If the application already exists, then selecting one of its fragments in the browser opens a textual editor on the application code. Figure 2 shows configuration browsing in progress for the Calculator application.

The current implementation of the fragment browser is based on endows with scrolling lists. Double clicking an entry from the list causes a textual editor to be opened on the corresponding fragment. But the fragment browser in its final form will resemble the graphical object browser of CLOS (Nørmark, 1991) or the hypertext browser of NoteCards (Halasz et al., 1987). Each fragment will appear as an icon in a browser window connected by arrows showing their relations. Clicking on an icon will bring up the corresponding fragment's code.

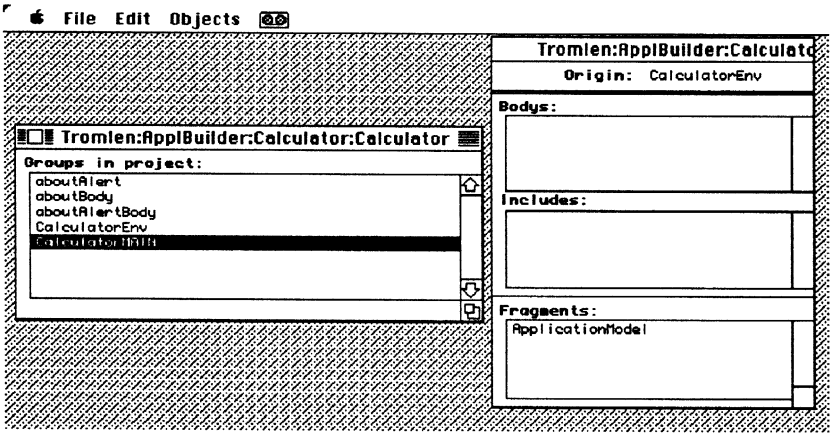

Figure 2: The fragment browser used for configuration editing. The leftmost window shows the minimal set of fragment groups for an ApplBuilder application under development. The rightmost window shows a list of properties for the selected fragment group, such as the latent groups being included, and the fragments in the group.

3.2 Graphical Editing of User Interface Objects

ApplBuilder provides graphical editors for basic UIO types like menu, endow, and dialog (alert). These graphical editors can be invoked from the textual editors either to add a new UIO or to inspect an existing UIO. Figures 3-5 show how a graphical dialog editor is used to build a new dialog for a Calculator. The editor is invoked from a textual editor open on the main program of the application.

If the name of a UIO is selected in a textual editor the appropriate graphical editor can be opened on its declaration similar to link-following in hypertext. (In future implementations we will provide textual editing of the UIO although the default editor will remain graphical.) In the graphical editor it is possible to edit the location and image of the UIO and its contents as well as small pieces of code associated with the UIO, called *scripts*, which capture the UIO's functionality.

To edit a UIO declaration graphically, its AST is interpreted to generate a graphical image. Information concerning the image of the object is represented twice: in the application code to be compiled, and in properties

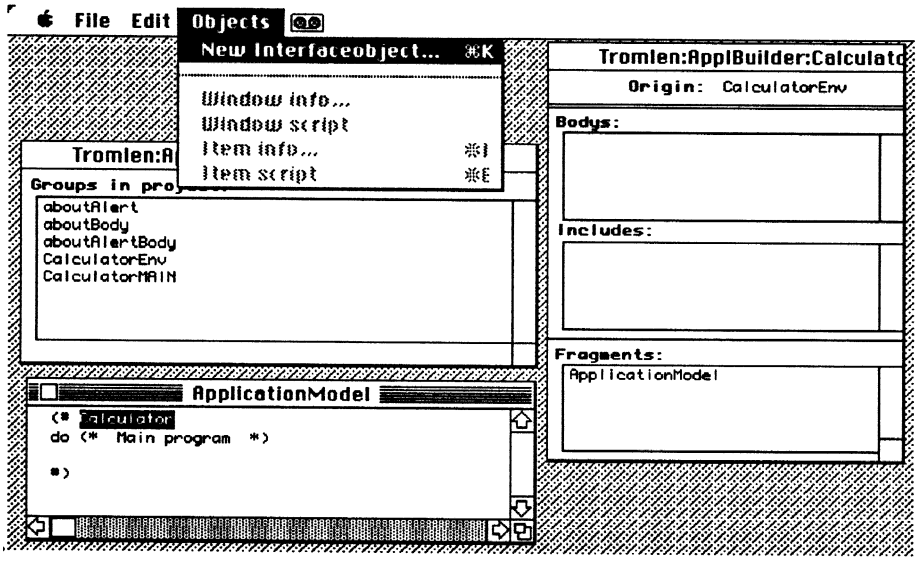

Figure 3: Adding the Calculator User Interface Object to the main program fragment.

attached to the top node of the AST representing the UIO. The properties in an AST are only visible to ApplBuilder; when compiling or textually pretty-printing the application the AST properties are ignored. There are, however, reasons for this duplication: (1) The graphical editor needs static information to display the objects and this information can easily be stored in properties. (2) It is impossible in general to gather image information from the code because we allow textual modification of the generated code. If the information on the location and image of the UIO on the screen is only present in the application code it is necessary to interpret the code for the object to be pretty-printed in the graphical editor. For example, displaying the image of a UIO may involve dynamic computations if its location and size depend on screen dimensions at runtime. In that case, static image information about the location and the size is not present at all in the code. (3) Reading the properties is also much quicker than scanning the AST for the specific imormation. When a UIO is to be displayed the properties are read and interpreted by the graphical pretty-printer and the object displayed accordingly. Even if some of the image and location information is to be computed dynamically in the application code the properties always hold static information on all images and locations. Thus, newly opened UIOs appear

the way they appeared during the last capital editing session. To make parts of a UIO be computed dynamically, the UIO's initialization routines must be edited textually.

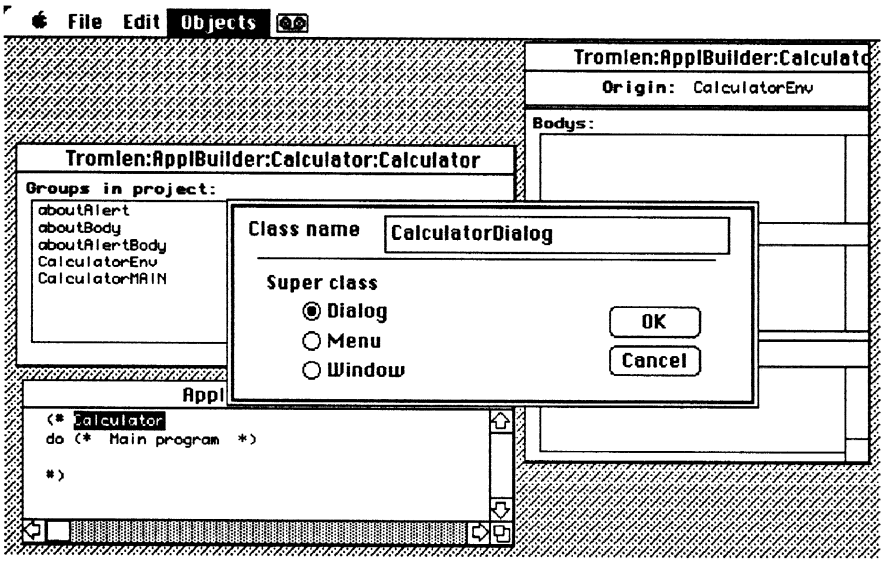

Figure 4: A new CalculatorDialog class inheriting from the general dialog class is added as the class description for the Calculator object.

The Dialog Editor

An example of a graphical UIO editor is the dialog editor. Items in the dialog editor window can be placed, moved, resized, and renamed by direct manipulation just as in Prototyper^{TM}'s or HyperCard's editors.

Figure 6 shows how a new item is added to a dialog and which properties of the item can be edited directly from the graphical editor. Similar to HyperCard each item is associated with a script describing its functionality. Such scripts can be accessed with a textual editor directly from the properties dialog as shown in Figure 6.[7]

[7]This dialog is invoked the first time through a pop up menu activated from the window background and later by double clicking the item in the editor window.

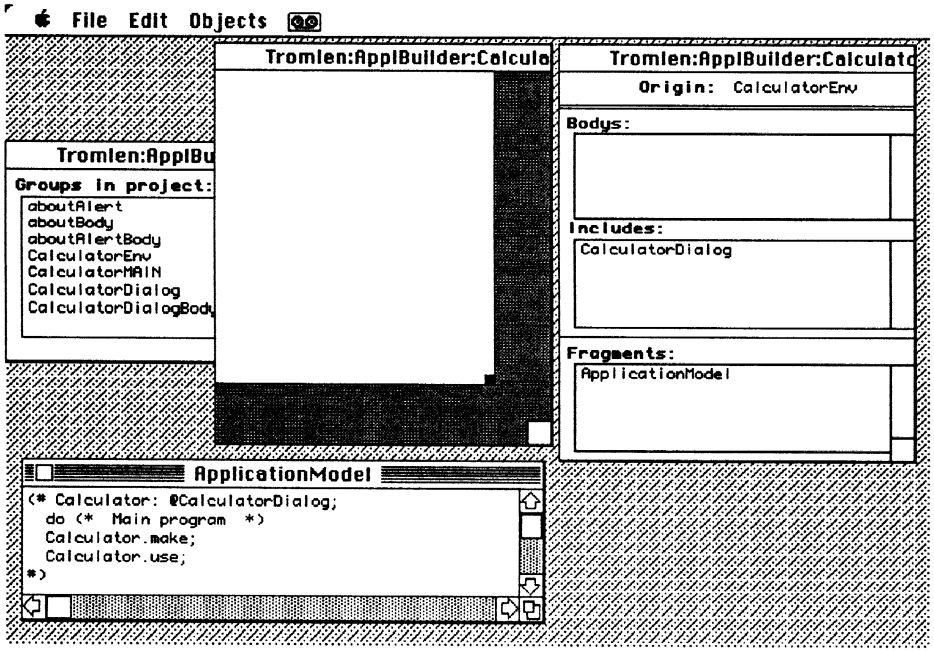

Figure 5: Adding a new dialog to a program fragment implies the generation of new fragments and the opening of a graphical dialog editor on an empty dialog window. Note that code for instantiating (': @CalcualtorDialog;'), initializing ('Calculator.make;', 'Calculator.use;'), and including relevant fragments (see leftmost window) are automatically inserted in the application code.

The dialog as a whole also has a script which can be invoked from the 'Objects' menu when the editor window is selected. In Figure 7 the window entitled 'Attributes' shows an excerpt of the script for the CalculatorDialog. In this script three classes of keys for the calculator are declared ('key', 'DyadFkey' and 'MonFKey'). Other script editor windows provide examples of scripts for instances of these classes. Currently we only provide textual editing for the item classes but in future implementations we expect to provide a graphical palette for editing both the class and its instances. It is possible to make specialization hierarchies of both UIOs and items that are either local to a dialog (cf. Figure 7) or generally accessible to many UIOs.

The graphical dialog editor window can be made larger than the actual dialog area, and items can be placed in this extra space during edit time (see

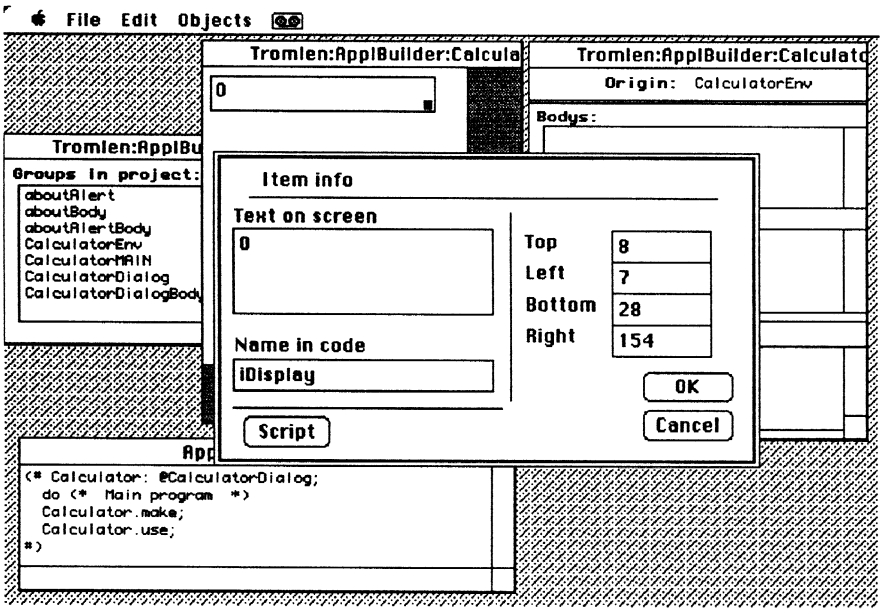

Figure 6: Adding the display item to the CalculatorDialog.

the 'New' button in Figure 7). In any case, all items in the editor endow are saved along with the dialog. The items in the work-space outside the dialog area cannot be seen or interacted with directly at runtime, but they can be called from the program. If the window size of the dialog is calculated dyamically, then these items may indeed appear in the window in some cases. This facility could be particularly useful in prototyping where designers could move items temporarily out of a dialog but still have them available for later reuse. The scripts of these "outer" items are in the same scope as the rest of the items, and hence are directly usable inside the dialog.

3.3 Textual editing

The application code, all written in the BETA language, can be edited textually in several ways. From the fragment browser textual editors can be invoked on arbitrary code fragments regardless of whether they can also be edited by graphical UIO editors. The BETA code is pretty-printed from the AST and displayed in a text editor window providing the usual text editing facilities. When the code from an editor window is saved it is automati-

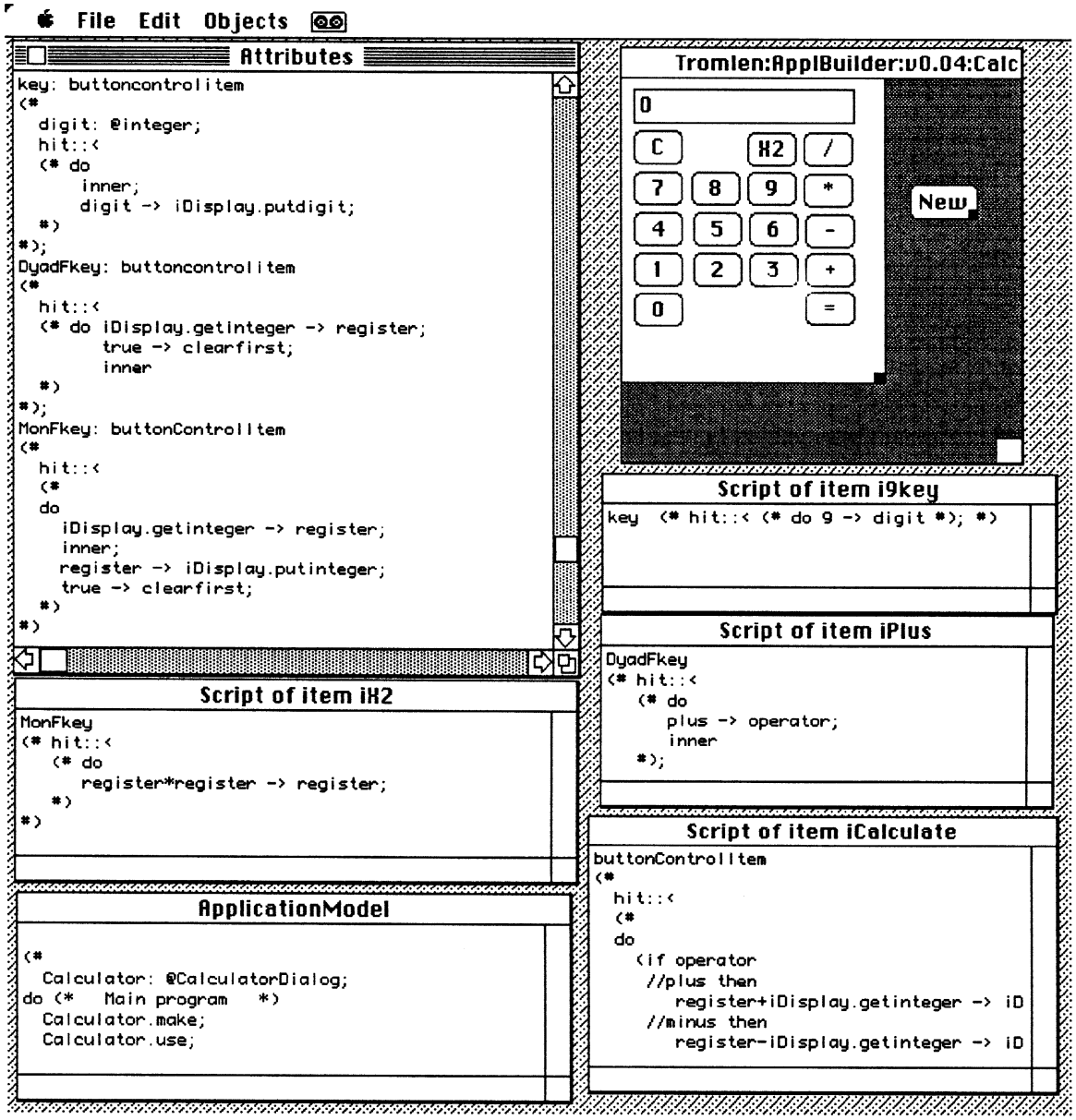

Figure 7: Editing 'CalculatorDialog' both graphically and textually. The graphical editor supports editing the placement of buttons, fields, etc. Textual editing of the dialog is done by invoking script editors on the items or on the dialog itself.

cally parsed and inserted into the AST. Multiple text editor windows can be open simultaneously displaying various parts of the same code as well as the contents of separate fragments.

From the graphical UIO editors, textual editors can be invoked on the scripts of the UIO, for example, on buttons and fields of a dialog. Any UIO has a number of scripts connected. A button has for instance a script allowing editing of local declarations and methods. Any script of a UIO may be opened and edited both from the fragment browser and from the UIO's graphical editor.

4 The Structure of Generated Applications

In applications where some of the code is automatically generated by the system it is important to keep the generated code and the code written manually apart. Furthermore, in order to support prototyping well it is necessary to provide maximum separate compilation of pieces of the code that change frequently during development. For example, scripts behind UIOs can be changed without re-compilation of any other part of the application code. This implies certain constraints on the physical organization of applications generated by ApplBuilder.

As shown in Figure 8, an application is divided into three parts: *environment, model* and *user interface*. The environment part provides access to pre-defined system classes and couples the model to the user interface parts. The model part contains fragments constituting the object-oriented "model" of real-world objects and concepts. The fragments constituting the user interface part handle interaction with the user and include any UIOs required by the application. Each UIO in the user interface part consists of a *main fragment*, containing declarations visible from the outside (the "interface" to the UIO), and one or more *implementation fragments* containing declarations of UIO items and scripts and the like which are not intended to be visible to the outside. This allows the contents of UIOs to be changed without re-compiling patents that use the UIO.

As an example, consider the dialog UIO, GetIntegerDialog, shown in Figure 9. Among the attributes visible externally are two operations, *use* and *make*. The *private* attribute is implemented in a separate implementation

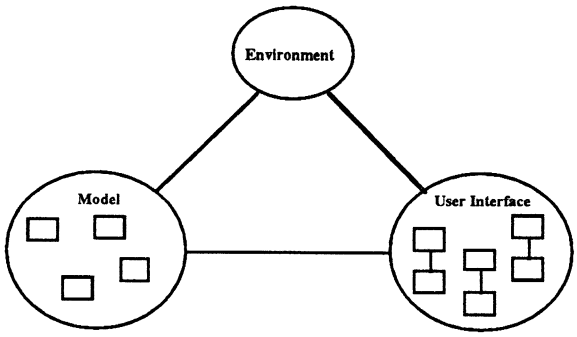

Figure 8: Structure of a generated application.

fragment and includes declarations of objects like the OK and Cancel buttons not intended to be visible outside the UIO. These declarations are generated automatically from the graphical editor while the actions, or *scripts*, of the two button items are written by the designer using a textual editor. The make operation also has a separate implementation fragment which includes code capturing the locations and sizes of the dialog items.

Designers can choose to start developing either in the model or the user interface. Often this choice determines the way control is managed between UIO classes and the model: (1) the model is the active part, invoking operations on the UIO instances and transferring parameters via the UIO operations or (2) the UIO is the active part, invoking model operations directly.

Figure 10 shows examples of each control management approach for the case of GetIntegerDialog. In Figure 10a, the model invokes the dialog through the use operation. When the dialog is closed, results are retuned to the model via the use operation's exit parameters. In Figure 10b, the UIO takes over some of the control and activates operations in the model. Here, control rests in the script of the dialog's OK button. Because model operations need to be visible in the UIO script, changes to model fragments can force recompilation of UIO fragments.

The UIOs of generated applications are instances of standard UIO classes from a large library. Figure 11a shows an excerpt from the classification hierarchy "above" GetIntegerDialog. The WindowItem classification hierarchy in Figure 11b is a component of the Window class shown in Figure 11a.

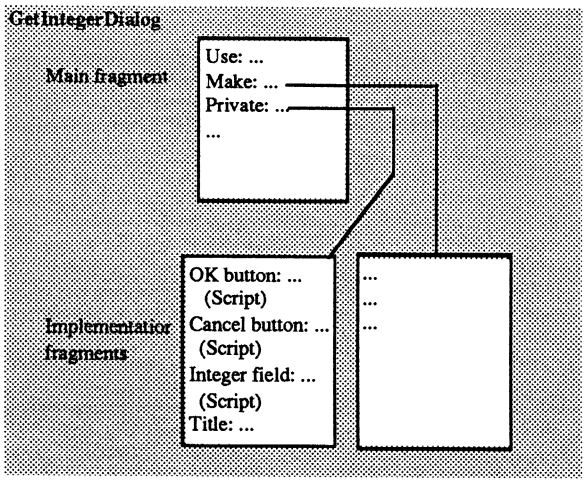

Figure 9: Structure of a dialog UIO.

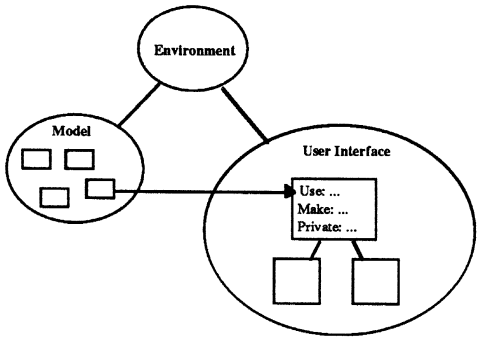

Figure 10a: Model invokes UIO

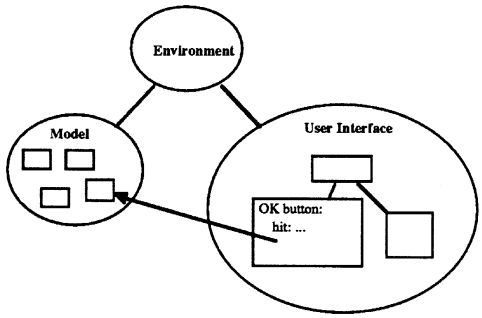

Figure 10b: UIO invokes model

5 ApplBuilder and Rapid Prototyping

ApplBuilder is a tool meant primarily for system desirers and developers, but as mentioned in the introduction, it is also flexible enough to apply in sessions where users and designers work cooperatively.

Because designers need to have freedom in their choice of approach to

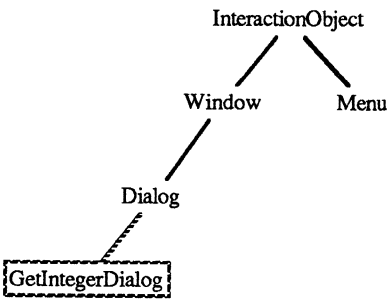

Figure 11a: Inheritance hierarchy for dialogs

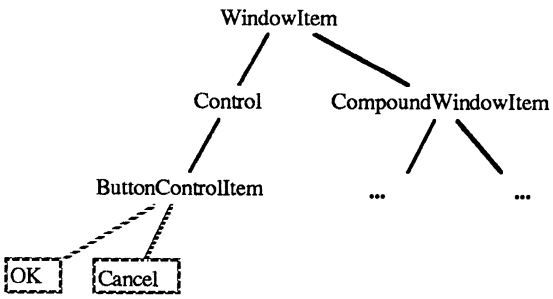

Figure 11b: Inheritance hierarchy for buttons

application prototyping and development, ApplBuilder does not enforce one particular style of development on its users. In fact, ApplBuilder supports at least the following approaches: (1) starting with design of UIOs and building up a horizontal prototype having only minimal functionality;[8] (2) starting from a "model", i.e. an implemented description of the functionality of a system, but with no UIOs implemented; (3) vertical prototyping, i.e. implementing the functionality behind a subset of a horizontal prototype or fully implementing a small subset of a system intended for incremental extension; (4) simulation of functionality, i.e. adding temporary short cut or dummy computations to a horizontal prototype to support sample data; and (5) full application development. In addition, these different ways of using ApplBuilder can be combined.

As described in the previous section, ApplBuilder modularizes application code so that a distinction is made between UIO modules and the modules constituting the object-oriented "model" of real-world objects and concepts. The latter correspond to real-world objects and concepts in a recognizable way. Furthermore, ApplBuilder deals with the application code in a form which makes it possible to save it, compile it and re-edit it in ApplBuilder.

Approach (1), horizontal prototyping, is supported by requesting ApplBuilder to start from scratch with a new application. In this case code

[8]Refer to (Floyd 1984) for an explanation of the concept of horizontal and vertical prototypes.

templates for a minimal application are generated, and the empty main program is opened in a textual editor. When the designer selects a new name in the declaration part of a program fragment, ApplBuilder can be requested to build a new UIO such as a window, a dialog, or a menu with that name (see Figures 3-5). Code to instantiate the UIO, include the fragments containing the UIO declaration, and the default initialization are automatically generated to be inserted in the fragment (see Figure 5).[9] A graphical editor is opened with an empty template for that particular kind of UIO. A hypertext-like *link* is established between the instantiation of the UIO and its declaration such that later traversal of the link opens up a capital editor on the UIO.

UIOs are designed by direct manipulation in a graphical editor and by writing scripts in a textual editor. Figure 7 shows an example of using a graphical editor to design a dialog. The scripts of the UIOs can also be used to link UIOs together. For example, the script of a menu item could bring up a dialog, pushing a button in a dialog could open a document window, etc. (This resembles the link feature of Prototyper.) This way interaction in a new application can be illustrated without programming the underlying model fragments. Apart from UIO scripts, the only code to be written is start-up code in the main model fragment. One can then compile and run the application as a horizontal prototype with little or no underlying functionality, and thus illustrate the visual appearance of the user interface.

Approach (2), starting from an application with no UIOs, is supported by opening ApplBuilder on the existing model application code. The textual editors invoked from the fragment browser can be used to inspect and modify the application code. In addition, the designer can start adding UIOs to the code just as in approach (1). It is also possible to open textual editors on application UIOs that were not generated in ApplBuilder, but the graphical editors cannot be used on such UIOs. As an example, consider porting a BETA program from one machine or window system to another. If the core functionality of the program to port is isolated in a number of model fragments, these can be moved to the new environment and the UIOs added with ApplBuilder.

Approach (3), vertical prototyping, is supported by the fact that the fine

[9]The designer can of course choose to insert the declarations in another fragment if needed.

grained modularization of the application code allows programming of functionality in fragments minimizing the need for re-compilation. For instance, parts of the functionality can be written in UIO scripts without touching and thus without re-compiling other parts of the code. A vertical prototype, for example, might have functionality behind only two of the menus appearing in the menu bar. UIO scripts could also include calls to particular model fragments such that re-compilation is restricted to such fragments when modifying the vertical prototype.

Approach (4), simulation of functionality behind UIOs is also possible. The local scripts of the UIOs can be used to store and display sample data that is supposed to be entered and displayed through the UIO. In this way, one can simulate the storing of data in a remote database before actually building the model that handles the database. This is useful for exploratory prototyping where groups of designers and users envision a new system (Grønbæk, 1990).

Finally, AppleBuilder can support approach (5), full application development. AppleBuilder utilizes the AST representation of the application code which allows editing and composing manually written and automatically generated code dynamically throughout the development process. This is in contrast to application generators that can only generate textual code skeletons. And unlike HyperCard and Prototyper™ AppleBuilder is embedded in a general purpose programming environment, the Mjølner BETA System. It is thus possible to convert prototypes generated by AppleBuilder into efficient application programs. In addition, one can use AppleBuilder as an implementation accelerator without building starting from prototypes.

6 Concluding remarks

The design of AppleBuilder provides flexible support for prototyping and user interface design while being embedded in a general purpose object-oriented programming environment that also supports full application development. Compared to HyperCard we provide similar support for local UIO scripting, but in AppleBuilder the scripts are written in the BETA language. The fact that BETA is used in the scripts means that we do not have to move to a different environment to write more advanced functionality as is the case with the HyperTalk XCMD interface to C and Pascal. It

also means that editing scripts is done in a powerful programming language which supports ordinary concepts like block-structure, abstractions mechanisms, encapsulation etc. A script is then part of a block-structured program and not as in HyperTalk where everything is either local or global.

Compared to other application generators such as Prototyper™ and Interface Builder we provide similar support for generating UIO code, but in ApplBuilder the code is generated as ASTs. This means that the code can be managed with the textual and graphical editors of ApplBuilder throughout the development process. In contrast, Prototyper generates Pascal code as text files that can only be edited with plain text-editors from the target programming environment. The designer is forced to leave Prototyper once the UIO's skeleton code has been created, because the generated files can not be read back in to Prototyper.

In Interface Builder, a reverse engineering problem appears when manipulating existing classes. Although the class's interface module is transparently maintained through the graphical editor, the implementation module is generated as a code skeleton in which the programmer implements class methods. Modifying an existing class with the graphical editor forces an attempt to generate a new empty skeleton for the implementation module. Unless the programmer wants to overwrite the old implementation module with a new skeleton, she must abort the generation and manually copy that part of the old code which is still applicable into the new skeleton.

Although we claim that the ideas behind ApplBuilder go beyond most existing application generators, there are certain problems remaning to be solved. In ApplBuilder we have strived to separate automatically generated code from the code written by the designer and to ensure minimal re-compilation of scripts. (Supporting fast modifications to UIO scripts is crucial in the rapid prototyping process.) The result is that the physical organization of the application makes the paths to some attributes rather long. A solution is to provide a future version of the ApplBuilder with interactive help to insert paths.

A common problem with automatic generation of code is the naming of classes and objects. Generated names are often either meaningless to the designer or too long. Hence ApplBuilder makes designers name all classes and objects. This means the designer is in charge of all names in the application as well as the names of files, fragments etc. constituting the

application.

The Mjølner BETA System is still only an industry prototype and thus certain features like concurrency are not implemented. Other pars of the environment are still not fully optimized and therefore sometimes present problems for designers. The current performance of the BETA compiler is also a problem. Slow compilation makes it hard to properly support rapid prototyping in cooperation with end-users. But the BETA compiler is under ongoing optimization and we hope that future versions of BETA will support our visions of using ApplBuilder in sessions with end-users.

Status of the current prototype

ApplBuilder is currently implemented in a prototype versions (Figures 2-7 in this paper are screen snapshots from the running prototype.) From the prototype, one can edit all fragments textually, although the text editors do not yet support syntax-directed programming. UIOs such as windows, dialogs, and alerts can be edited graphically. However, window type, button highlighting, and the like are not visible from the editors. These features can only be inspected from the window property dialog and seen at runtime. It is in any case possible to generate a full set of BETA code skeletons for an application and then add model fragments and UIOs.

Future work

Future work on ApplBuilder will take place within the framework of a project called DeVise, concerned with developing and integrating tools and techniques for experimental system development (Grønbæk & Knudsen, 1991). In that context, certain features of the current ApplBuilder will be moved into independent modules called the *fragment browser, hypermedia link manager*, and *AST manager*. The remainder of ApplBuilder will become a module called the *user interface design tool*. In the integrated DeVise environment, user interface designers will have transparent access to services (in the form of other modules) not currently part of ApplBuilder, including syntax-directed editing and an object-oriented CASE tool for graphical editing of BETA programs (Sandvad, 1990).

Acknowledgements

We would like to thank Ole Lehrmann Madsen and Jonathan Grudin for their comments on earlier versions of this paper. We would also like to acknowledge Henry Michael Lassen for the effort he put into the programming of the first prototype of ApplBuilder. This work has been supported by *The Danish Natural Science Research Council*, grant no. 11-8385.

References

[1] Bak, L., Nørgaard, C, Sandvad, E., Knudsen, J.L., Madsen, O.L.: *An overview of the Mjølner BETA System.* Software Engineering Environments, Wales, March 1991.

[2] Bjerregaard, B.S, and Hviid, A, The Development of a Graphical Object-Oriented Prototyping System - GrOOPS. *Tech. Rept. 93, IR, Computer Science Department*, Aarhus University, Arhus, Master-thesis, May, 1990.

[3] Boar, B. H. *Application Prototyping - A requirements definition strategy for the 80s*, John Wiley and Sons, Inc., New York (1984).

[4] Bødker, S. & Grønbæk, K. Cooperative Prototyping Experiments - Users and Designers Envision a Dental Case Record System. In *John Bowers & Steve Benford (eds.) Proceedings of the first EC-CSCW '89*, UK, September 1989. Computer Sciences Company.

[5] Goodman, D. The Complete HyperCard Handbook, Bantam Books, New York (1987).

[6] Grønbæk, K. & Knudsen, J. L. Tools and Techniques for Experimental System Development. Draft submitted for publication. Computer Science Department, Aarhus University, September 1991.

[7] Grønbæk, K. *Prototyping and Active User Involvement in System Development: Towards a Cooperative Prototyping Approach.* Computer Science Department, Aarhus University, Ph.D. Thesis, January 1991.

[8] Grønbæk, K. Supporting Active User Involvement in Prototyping. *Scandinavian Journal of Information Systems* 2:(pp. 3-24), 1990.

[9] Halasz, F.G., Moran, T.P., and Trigg, R.H.: NoteCards in a Nutshell. In *Proceedings of the ACM Conference on Human Factors in Computer Systems* (CHI+GI '87) (Toronto, Ontarion, Apr. 5-9). 1987, pp 45-52.

[10] Kristensen, B.B., Madsen, B.B., Møller-Pedersen, B., and Nygaard, K: *Object-Oriented Programming in the Beta Programming Language.* Book in preparation, January 1990.

[11] Kristensen, B.B., Madsen, B.B., Møller-Pedersen, B., and Nygaard, K: The BETA Programming Language. In: *B. D. Shriver, P. Wegner(eds.), Research Directions in Object Oriented Programming*, MIT Press, 1987.

[12] Lantz, K. E. The Prototyping Methodology, Prentice Hall, Englewood Cliffs (1986).

[13] Madsen, O.L., Nørgaard, C.: An Object-Oriented Metaprogramming System. *Hawaii International Conference on System Sciences - 21*, pp 406-415, January 5-8,1988.

[14] MIA-90-3 *Mjølner Informatics Report*: The Mjølner BETA Fragment System - Reference Manual. Mjølner Informatics ApS, Science Park Aarhus, February 1990.

[15] NeXT reference manual. Chapter 8. Interface Builder.

[16] Nørmark, K.: A Hyperstructure Programming Environment for CLOS. *Fourth International Conference on Technology of Object-Oriented Languages and Systems.* (TOOLS '91) Paris, March 4-8, 1991.

[17] Sandvad, E. Hypertext in an Object-Oriented Programming Environment. In: *J. Andrew J. Bezivin (eds.): Woodman'89: Workshop on Object-Oriented Document Manipulation*, Rennes May 1989, BIGRE.

[18] Sandvad, E: Object-Oriented Development - Integrating Analysis, Design and Implementation. *DAIMI PB-302, Computer Science Deportment, Aarhus University*, April 1990.

[19] Shafer, D., *HyperTalk Programming*, Hayden Books, First edition, Second printing 1988.

[20] Squires, S. L., Branstad, M., and Zelkowitz, M. (eds) Special Issue on Rapid Prototyping. Tech. Rept. 5, *Software Engineering Notes, Vol 7., ACM SIGSOFT*, Baltimore, Working papers from ACM SIGSOFT Rapid Prototyping Workshop, April 1982.

[21] Wilson, J. & Rosenberg, D. Rapid Prototyping for User Interface Design. In Helander, M. (ed.) *Handbook of Human-Computer Interaction.* North-Holland, Amsterdam, 1988.