View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by INRIA a CCSD electronic archive server

A distributed architecture for programming environments

Dominique Clement

To cite this version:

Dominique Clement. A distributed architecture for programming environments. [Research Report] RR-1266, INRIA. 1990. inria-00075293

HAL Id: inria-00075293
https://hal.inria.fr/inria-00075293

Submitted on 24 May 2006

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

INRIA

UNITÉ DE RECHERCHE
INRIA-SOPHIA ANTIPOLIS

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P. 105
78153 Le Chesnay Cedex
France

Tél.:(1) 39 63 55 11

Rapports de Recherche

N° 1266

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

A DISTRIBUTED ARCHITECTURE FOR PROGRAMMING ENVIRONMENTS

Dominique CLEMENT

Juillet 1990

Une Architecture Répartie pour les Environnements de Programmation

A Distributed Architecture for Programming Environments

Dominique Clément

SEMA GROUP
c/o I.N.R.I.A. Sophia Antipolis
Route des Lucioles
06 560 Valbonne FRANCE
clement@mirsa.inria.fr

Résumé. Les environnements de programmation sont basés sur des concepts tels que la syntaxe et la sémantique, et ils fournissent des services tels que l'analyse syntaxique, l'édition, la vérification de typage, et la compilation. Mais la plupart des environnements de programmation existants actuellement sont organisés autour d'une architecture trés integrée, dans laquelle les composants sont fortement couplés. Cela conduit à des systèmes qui sont la somme de leur éléments, avec des implications évidentes en termes de taille, de réutilisabilitée, et de maintenance. Dans ce rapport, nous proposons une architecture répartie pour les environnements de programmation.

Mots clés: Environnements de programmation, Intégration des données, Intégration du contrôle, interface utilisateur, le système Centaur.

Financement: Ce travail est partiellement financé par la Communauté Economique Européenne, projet n° 2177.

Abstract. Programming environments are typically based on concepts, such as syntax and semantics, and they provide functionalities, such as parsing, editing, type-checking, and compilation. But most existing programming environments are designed in a rather integrated manner, where parsers, editors, and semantic tools are tightly coupled. This leads to systems that are the sum of all their components, with obvious implications in terms of size, of reusability, and of maintainability. In this paper, we present a proposal for a distributed architecture for programming environments.

Keywords: Programming environment, Data integration, Control integration, User Interface, the Centaur system.

Grant: This research is partially supported by the European Economic Community, under ESPRIT Project number 2177.

A Distributed Architecture for Programming Environments

Dominique Clément
SEMA-GROUP Sophia-Antipolis

Abstract

Programming environments are typically based on concepts, such as syntax and semantics, and they provide functionalities, such as parsing, editing, type-checking, and compilation. But most existing programming environments are designed in a rather integrated manner, where parsers, editors, and semantic tools are tightly coupled. This leads to systems that are the sum of all their components, with obvious implications in terms of size, of reusability, and of maintainability. In this paper, we present a proposal for a distributed architecture for programming environments.

1 Introduction

Tool integration has become widely recognized as a central issue for the future of software development environments. It allows the cooperation of many tools, from many sources, in a uniform but open framework. Some systems organized around this principle have recently emerged [7], [20], [28]. Quoting the HP CASEdge Soft Bench, it is possible to identify three key aspects of tool integration: data, control, and user interface.

Data integration addresses the sharing and exchange of data between tools. This is usually achieved with structures defined as data models that tools can interpret. Control integration is more related to communication issues between tools. Finally, user interface integration is needed to provide a uniform appearance and interaction model.

In the field of software development environments, these aspects are tackled respectively with techniques such as full data integration in the spirit of PCTE [6], message based communication architecture in the spirit of the Field Environment [26], and graphic interactors with a standardized "look and

feel", for example Motif [25]. The use of distributed computing environments implies that the tools execute remotely.

Programming environments are still designed in a way such that the tools are tightly coupled. These systems are based on well-defined concepts, such as the syntax and the semantics of programming languages, and provide similar functionalities. However, tool integration is mainly achieved by the use of a common memory storage.

We propose to organize programming environments as sets of cooperative tools; this being based on tool integration. Here we take advantage of the specificity of programming environments to achieve a more complete tool integration. We group the components of a programming environment within three different classes: *syntax*, *editing*, and *semantics*. Tool integration can be defined in terms of data and control integration between classes and a user interface generated by a toolbox.

We describe our architecture in three steps: data integration, control integration, and user interface integration. We start with data integration based on our three classes. Control integration is then described using an encapsulation technique. Finally we present a toolbox approach for user interface integration. We conclude with a discussion of related work.

2 Data Integration

2.1 The Syntax Class

Today, most programming environments include a kernel for syntactic processing, i.e., creation and manipulation of structured objects. In particular, most are based on the notion of *abstract syntax* of a programming language. The environment's objects are the *trees* generated by a given abstract syntax. Note that sometimes *graphs* are used rather than trees.

Data integration in programming environments is primarily based on abstract syntax trees: all the tools are cooperating by using the same abstract representation. This property is used by some syntactic kernels to define syntactic objects as abstract data types.

Most syntactic kernels provide navigation, modification, pattern-matching, and annotation facilities. Also a persistent storage mechanism usually exists. However, none of existing programming environments share the same kernel. This can be attributed mostly to the use of an integrated architecture. The integration of technological changes is thus very difficult.

We propose to structure the syntactic kernel as a *server* which provides services that are solely syntax-oriented, such as the creation and the management of structured data. All other tools, e.g., parsers, evaluators, editors, etc.,

have access to abstract syntax objects through the services provided by the server.

A Syntactic Server

The syntactic server provides the primitives to define an abstract syntax, typically in terms of *sorts* and *functions*. Such an abstract syntax object contains all the necessary information to create abstract syntax trees and to perform any syntactic checking. The syntactic server provides primitives to create trees. Clearly, all the trees created with a given abstract syntax are stored within the server's memory. Trees are known to the tools only through a unique tree identifier.

Of course, it is no longer possible for a tool, now being a client, to manipulate the trees directly. But tools still have to explore and modify trees. A standard approach is to define an abstract interface which defines the primitives to navigate in a tree (e.g., *down*, *left*, *right*, *up*, etc.), to modify a tree (e.g., replacing a subtree by another one, inserting and deleting elements in lists, etc.), and to reference locations. Tools manipulate trees using these abstract primitives. The main question concerns the style of these primitives.

One possibility is to use an applicative approach with tree primitives working on tree identifiers. While such a solution may work for tree navigation, it raises difficulties when modifying a tree or referencing locations. Although trees are only known through tree identifiers by clients, the syntactic server itself has to manage possible concurrent access. A typical example is a tool that references a subtree which another tool is modifying. What happens to the tree identifier known by the first tool after the second tool has modified the tree? The only information the server could maintain is a list of "out-of-date" tree identifiers.

We propose to use an imperative language based on the notion of tree *occurrences*. Occurrence is a well-known notion [17] used to denote tree addresses as lists of positive integers. For example, occurrence [2; 1] denotes the 1st subtree of the 2nd subtree of a given tree.

Of course, occurrences can be used to memorize locations of interest, for example all places where some external data has been attached to a tree. Moreover, they can be used to navigate in a tree and to modify a tree. Navigation, for example, can be done as follows: given some occurrence u which leads to the subtree S, one can go to the nth son of S by adding the number n to the end of u. Modifications can be specified as: replacing the subtree at occurrence u within a tree T by a tree N.

The syntactic server can provide operations on occurrences. One such operation is the *at* function, which applies an occurrence u to a tree T resulting in a subtree S: $S = T\,at\,u$. Here, no subtree is returned, but a query can be

sent to the server using this operation, e.g., what is the top operator *at* the occurrence u.

If we define a *variable* as a pair of a tree and a set of occurrences, client tools only have to access variable identifiers. From a variable, a tool manipulates a tree with respect to an occurrence identifier, using abstract primitives on occurrences.

Occurrences are created from client requests, but they are memorized within the server. Thus, the server is responsible for the management of the occurrences within a tree. In particular, it makes sense for the server to have specialized updating primitives on occurrences to keep them consistent with respect to tree modifications. For example, if a tree replacement occurs at an occurrence u which is "above" an occurrence v, i.e., $v = u.u'$, the occurrence v is no longer valid. The server could modify the occurrence v, i.e., $v = u$, and notify the interested clients.

The advantages of the imperative solution over the applicative one are to enable the server: i) to keep a precise and updated list of all the occurrences within a tree and ii) to report the tree modifications that have occurred since the previous usage of a tool. Thus, tools register with the server the first time they work on a given variable. For each such registered tool, the server memorizes the locations in the corresponding tree where modifications have occurred.

2.2 The Editing Class

Programming environments provide language-specific editing facilities. These usually extend textual editing operations with structure-oriented editing. Typically the environment editor generates concrete representations from the abstract structures. Structured editing is based on a mapping between textual positions within the concrete representation and tree addresses. This mapping is used in two directions: from abstract to concrete to compute the concrete representation, i.e., the "display area", which corresponds to a given sub-object; from concrete to abstract to compute the sub-object which corresponds to a position, i.e., the selection.

In programming environments, the mapping between abstract and concrete representations is usually achieved by a projection function, also called a pretty-printer [29], [24]. Some environment editors use a parser instead of a pretty-printer. Here, the parser constructs the necessary references by using the tokens positions [11].

Most existing programming environments provide their own specialized editor, which is tightly coupled with a parser and a pretty-printer. Very often such editors have less textual capabilities than full textual editors.

Note that structured editing is a special case of connecting different representations using a selection mechanism. Examples include specialized editors, e.g., a graphical editor for graphical objects, and interactive tools, e.g., type-checkers and symbolic debuggers. In the latter case, the selection mechanism relates information coming from an interactive tool to the data that tool is using.

We propose to restrict the editing class to the structures and tools that are needed to implement the mapping between abstract and concrete representations. Other structures, in particular those needed for syntax checking and those needed for editing, should not be part of the editing class. This leads to an editing architecture based on data integration with a formatter.

Editing Architecture

The editing architecture is organized around a central component, a formatter. The formatter uses a document description language as input and generates an image structure as output. The image structure, for example a tree of graphic boxes, contains all the necessary display information such as size, font, color, etc. We connect such a formatter to other components, e.g., a syntactic server and an editor.

The coupling with an editor is based on the exchange of position information with respect to the image structure. The formatter must provide a selection mechanism to transform a position (X, Y), given by the editor, into the proper image. Moreover, the formatter must be able to compute the "display area" of the selected image, which can be given back to the editor to be highlighted. This could be, for example, a rectilinear area. Here, a selection corresponds to a "path" within the image structure. In the remainder of this section, we consider that such a path is represented as an occurrence.

The coupling with a syntactic server is based on the exchange of occurrences. We could assume that there is a one-to-one correspondence between occurrences within the server's data and occurrences within the image structure. More generally, we can assume that the programming environment provides a translator between occurrences. Such a translator is derived from a description of the mapping of abstract schema to image structure. This description uses, of course, the formatter's document description language. Thus, syntax information on the selected image can be obtained from an image occurrence by translating it into a tree occurrence and querying the syntactic server.

To compute a structured "display area", the formatter must find the proper set of elements in the image structure. One possibility is to have the formatter's document description language include selection commands one can use to wrap a selection image around graphical images. A structured selection is obtained by walking up the image structure to a selection image. Another

possibility is to use the translator between occurrences: an image occurrence is given to the translator which computes an occurrence within the corresponding tree. Note that in certain cases the image occurrence must be modified (this corresponds to walking up the image structure).

The above described model is depicted in the figure 1. Of course, the communication is bi-directional.

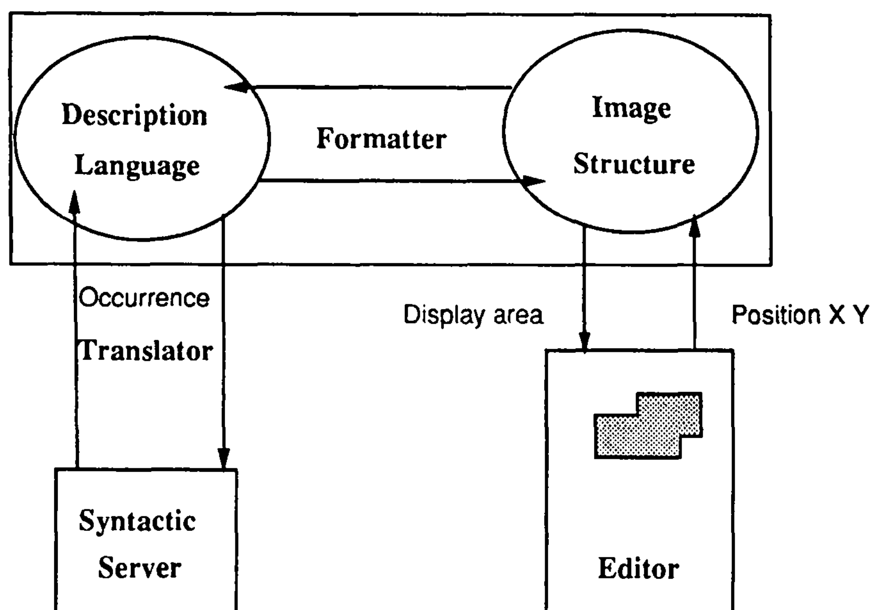

Figure 1: Editor Architecture

To complete our editing architecture, we must also explain how modifications are handled. Here, we separate modifications issued from the syntactic server from those issued from the editor. That is, structure editing from textual editing.

For the former, when a tree is modified the formatter updates its image structure, using modification information passed by the server, and then informs the editor of the changes. Of course, it is preferable to have an incremental update mechanism.

For the latter, we take advantage of our architecture to limit editing operations to the editor and the formatter as long as no syntactical checking is required. The formatter must maintain the image structure after textual modification: when editing occurs within a display area the editor gives geometric information to the formatter, which updates the image structure. It is only on a specific request that the correspondence between the image and the tree structure is updated.

Note that the formatter and the editor do not have to be specific to the programming environment. Moreover, their software evolution are independent.

2.3 The Semantic Class

Programming environments include semantic components such as type-checkers, interpreters, debuggers, etc. There are two subclasses of tools:

- Those with their own internal data representation. For such tools we need a coercion protocol between the tool data and the server data.

- Those using the server's trees as an opaque data structure they access through a predefined interface. No coercion is needed, but the tool must connect to the server often.

Data integration of the semantic class is based on the use of the abstract interface provided by the syntactic server. This implies that data integration is limited to those tools whose data can be modeled with the data abstraction provided by the server. This is fulfilled by tools of the second subclass, although not necessarily by those of the first.

We want to illustrate semantic data integration for semantic processors based on computation of equations driven by tree traversal. Although the tree traversal could be coded recursively, we prefer an imperative manipulation on occurrences. The computation is done using information the tool obtains from the server, according to the current state of a tracing occurrence. It is then possible for the tool to use the server's occurrence facility to memorize locations, e.g., places where some error has occurred.

Attribute grammar based tools are typically "traversal driven". The tool specification is written in terms of an abstract syntax description and equations. An attribute grammar generator transforms the specification into a table driven program which does almost all the memory management and computation of the semantic values. Of course, the transitions of the program depend on the given input tree. Here, we propose to replace the usual recursive implementation technique by imperative navigation commands using occurrences.

3 Control Integration

Programming environment components need to communicate. For example, the editor may be coupled with the type-checker to provide semantic assistance to editing operations; the debugger is coupled to the evaluator. One may want

to interface the error messages generated by a compiler so that the user can correct a program. Or perhaps, one wants an interpreter to display the current values of variables during execution.

Intertool communication is a very general issue within software systems. Some recent work indicates that a good approach to achieve tool integration is to have the control of information external to the tools. Hypertext systems are more or less based on this principle: communication is expressed by external scripts. Some programming environments are organized around a distributed architecture using message-based communication techniques [26], [9], [16]. But control integration is not limited to communication between tools. It is also a matter of concern within a single tool.

3.1 An Object Oriented Approach

One possibility for communication within a tool is an object-oriented approach. In object-oriented architectures, message passing provides modularity and abstraction. Using message passing one can write generic programs. Furthermore, when writing the code one does not have to discriminate over all possible types. It is only at application time that the choice of the appropriate code is computed from the type of the values. For a value to be a valid parameter for a generic program it must belong to the type hierarchy which provides the code. Such code is called a "method" and one says that objects respond to methods.

When dealing with communication, the standard message passing technique appears to be insufficient. Consider the coupling of a scrollbar and an object that one wants to scroll over. Clearly the scrollbar must be able to communicate with the object, e.g., to send a "scroll" method to that object. Here message passing seems to provide the necessary independence between the scrollbar and the scrolled object, since one can argue that the scrollbar must know how to find the object. The situation is less clear if the object can scroll by itself, e.g., a text editor. In that case the object must be able to communicate with the scrollbar. The question is how to establish that communication. That is, must the object know if there is a scrollbar or not.

A first solution is to use the existing object structure as a vehicule for messages. In our example, the object could send a "scrollbar" message that propagates over the structure in which the object and the scrollbar are combined. This approach presents at least two deficiencies. First, it requires a common data structure "linking" several objects. Second, the program that implements the scrolling primitive of the object must test for the scrollbar's presence. Thus, we use a more distributed message passing technology where messages are emitted without knowledge of receivers.

3.2 Distributed Message Passing

We propose to organize control integration between objects using an event programming technique: objects communicate by sending and receiving events.

Consider the case of a type-checker which reports errors. To integrate this type-checker within an interactive environment, we need a mechanism to couple the type-checker output, i.e., the errors, with a browsing facility. Here, we are concerned with the communication between a source window and an error window. For example, after running the type-checker on a program perhaps the list of errors and warnings appears in a separate window on the screen. When moving from error to error, by selecting the error of interest, one has a selection in the source window. Conversely, from the source window, one may ask for the error message corresponding to a given selection in the source.

Note that at the time the error window is coupled with the source window, it is likely that the latter is already connected to other objects. Of course, we do not want such connections modified. Also this new connection should not interfer with existing ones. The control integration mechanism must be dynamic.

Our approach is to use the notion of observers familiar in concurrent languages [23]. The capability of an object to communicate with the external world is characterized by a set of inputs and outputs. Inputs correspond to the primitives provided by the object. Outputs correspond to the messages, we call them signals, the object can emit. (Note that signals can carry a value.) Inter-object communication is then described as a network of connections between objects.

We define a *node* type by a *name*, *input* ports, and *output* ports. This corresponds to a module interface declaration. A node instance can be associated with an object, provided that it encapsulates the object's external interface. For such node instances, which we call *atomic*, inputs are associated to the appropriate object methods. The object sends messages by activating the output ports of its associated node. Thus, it is necessary to maintain an association list between objects and atomic nodes.

Communication between nodes is achieved by creating a *network*. Within a network, the output ports of a node are connected to the input ports of other nodes, provided that the ports have the same name (this restriction can be relaxed using port renaming). The output port of a node can be connected to the input ports of several nodes, i.e., there is broadcasting of signals. In principle, the input port of a node can be connected to the output ports of several nodes.

A network has an interface with the external world, i.e., it is itself a node.

The input ports of the network are connected to the corresponding input ports of the encapsulated nodes, while the output ports of these nodes are connected to the network's output ports.

Communication between nodes of a network is synchronous but not instantaneous. For example, if a node has an input port connected to several output ports, the node receives several signals through its input port, each one with its own value. The network is responsible, if necessary, for defining its own clock. Except for atomic nodes, the behavior of a node on receiving a signal on a port is to broadcast that signal to all the nodes that are connected to it. Note that the signals that are neither input nor output ports of the network are local signals, whose communication is limited to within the network.

An example

We illustrate our communication method on the connection between a source window and an error window. To simplify, assume that we have only one node type, say `view`, with a `selection` input and a `selection` output. Initially, the source window is associated with an instance of a `view` node which is connected to other nodes within a network. To couple the error window with the source window, one only has to associate the error window to another instance of a `view` node and to include this node within the network. This establishes all the necessary connections.

Every time the error window emits a `selection` signal, typically the result of some user action, e.g., pointing at an error message with the mouse, the source window receives that signal through its `selection` input port. The obvious semantics of the method attached to that input port is to move the selection mechanism of the window to the proper location in the source code. Conversely, every time the source window emits a `selection` signal through its output port, the error window receives that signal. (Here the value of the `selection` signals can be an occurrence identifier.)

3.3 An Event Handler

For atomic nodes, we want the communication with their associated objects to be asynchronous. We briefly describe the principles of an event handler.

- The event handler maintains the association list between objects and nodes.

- An object can emit an event using the "emit" primitive:

```
emit <object-id> <port-id> <value>
```

where the `port-id` corresponds to an output port of the node associated with the `object-id`. When this function is called, an event is created which is given to the event handler, e.g., added to a queue of events.

- When the event is read by the event handler, the `object-id` is used to find the associated node using the association list. Then the `port-id` signal is sent to the set of nodes that are interested by the message, i.e., all nodes that are connected to the emitting node. The `value` is not examined by the event handler.

- All connected atomic nodes are activated, i.e., the function associated with their input port is called. Here it is necessary for a node to have access to its associated object, because the action to be executed belongs to the object address space.

This description assumes that all the objects are in a single tool. In the context of a distributed architecture it is also necessary to have a mechanism to connect different tools. This kind of intertool communication can be achieved using a broadcast message server as in [26], [7].

4 User Interface Integration

The user of a programming environment manipulates structures and runs special processors and translators. This is typically achieved with a user interface *via* a personal workstation, a keyboard, a mouse, and a high resolution screen. The way a user interacts with a programming environment generally depends on the tasks that can be performed.

4.1 Design Principles

A general user interface design principle is to disconnect the man-machine interface from the system and to keep the man-machine interface open-ended. This corresponds to a decomposition into two components: i) the *external* interface, e.g., interactors such as buttons, menus, dialogs, etc.; ii) the *internal* interface, e.g., application specific editors. This is depicted in figure 2.

For the external interface, an extensible set of graphical interactors must be available. Many toolkits [25], [22], [1], [10] are good possible candidates. These are based on the same principles: use of the distributed X-window system [15], a reasonable sampling of widgets, and extensibility. Here, we believe it is important for the external interface to present a standardized appearance. Currently *Motif* provides such a standard.

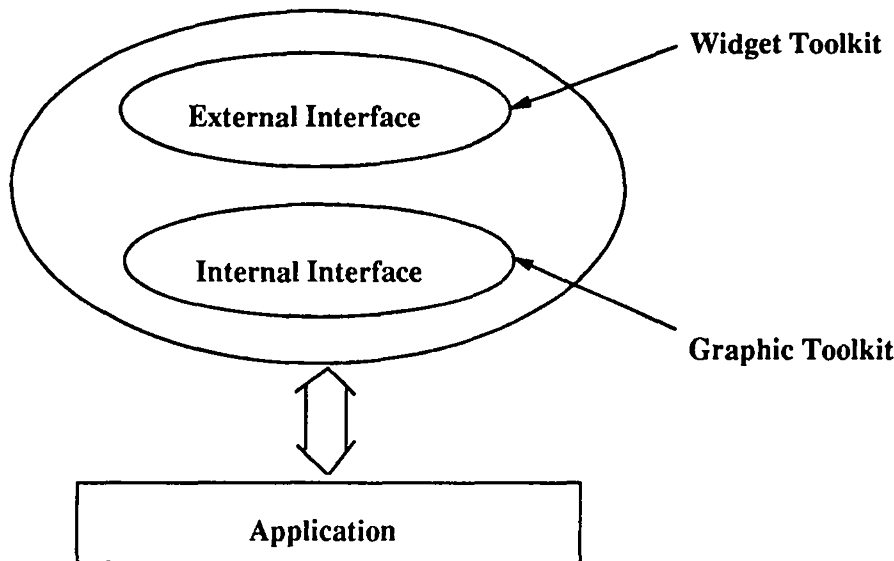

Figure 2: User Interface Components

For the internal interface, we need a higher level graphic toolkit to help display application-specific data structures, for instance the image generated by a formatter. This level must provide the necessary primitives to display a graphic, to select a graphic from a position, to modify the graphic's appearance (e.g., change its color, orientation, size, etc.), and to modify the graphic's structure. Finally, the graphic toolkit and the application must be connected to allow cross-referencing between graphic and application objects.

The definition of a general purpose graphic toolkit is still an open question. However, some systems have recently come into existence [22], [2], [16], [13]. Some document processing systems are evolving towards graphic toolkits [12]. We believe it is possible to define a general user interface toolbox to implement a programming environment's user interface.

4.2 A User Interface Toolbox

Work on user interface management systems, UIMS, has lead to a three module approach to defining the user interface of an application. The first module, the *application interface*, is responsible of the communication between the application and its interface. The second module, the *presentation manager*, is responsible of the *look* of the interface. The third module, the *dialog manager*, is responsible of the *feel* of the interface. This is shown in figure 3.

For programming environments, we propose to have the three modules generated with a user interface toolbox. Such a toolbox should provide the

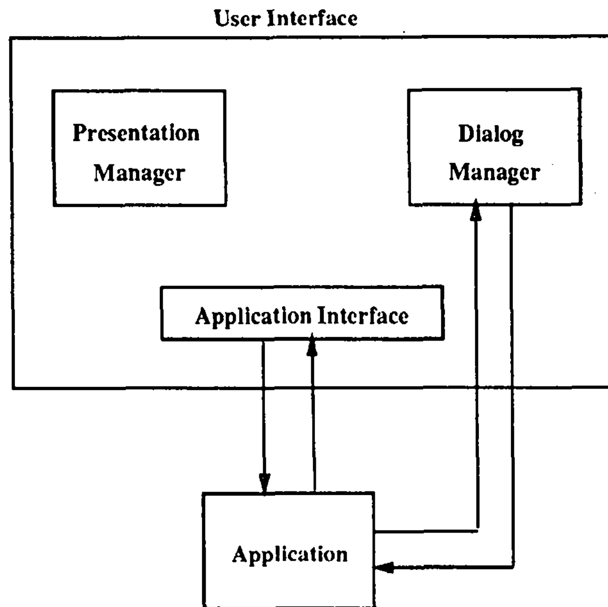

Figure 3: User Interface Management System

following functionalities:

1. A data integration mechanism that is used to generate the application interface.

2. A user interface description mechanism that is used to generate the presentation manager.

3. A control integration mechanism that is used to generate the dialog manager.

Before going into further details of the toolbox, we describe the architecture of the user interface that we want to generate.

Application Interface

One possibility for the application interface is to use shared variables between the application and the user interface, [27]. But this makes it difficult to manage software evolution of both the application and the user interface. Also it only works in a distributed context if there is a common memory.

We follow an approach similar to the Chiron system [3], where the application is separated from the user interface by an image processor. This image processor is responsible for the management of the concrete representation

associated to application data. In our context, this image processor can be a formatter coupled with a syntactic server, as explain in the Editing Class subsection. As a consequence, the application interface can be reduced to the syntactic server protocol. The application only has to generate commands that can be interpreted by the server. The visual effect of these commands depends heavily on the graphical capabilities of the image processor.

Presentation Manager

Here, we believe it is important that the presentation manager to be application independent. To achieve this, we propose to use a control integration mechanism between an image processor and a graphical user interface. By graphical user interface we mean the coupling of external and internal interface components. The image processor contains the necessary information for the internal interface component to display the image structures. The graphical user interface is solely responsible of the management of the appearance of interface.

Of course, there is bi-directional communication between the image processor and the graphical user interface. This is depicted in figure 4.

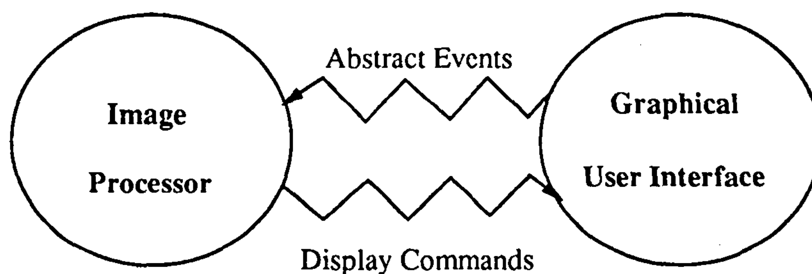

Figure 4: Connection between image processor and a graphical user interface

Dialog Manager

It should be clear from the described architecture that the application is completely isolated from any graphical operation of the user interface. However, the application and its user interface are not only connected through the application interface: the application may send commands to the user interface, e.g., creating new widgets; the user interface may send commands to the application, e.g., calling some function. It is precisely the role of the dialog manager to establish such communication between the application and the user interface.

We feel the user interface must be responsible of the management of the user inputs. For example, with Motif, this is achieved by the main event loop, *XtMainLoop*. Also, the user interface must only communicate with the application through an abstract event mechanism, that is independent of application semantics. For example, an *activated* event is always generated in reaction to user actions, e.g., releasing the mouse in a push-button widget independent of the button's semantics. The dialog manager interprets the abstract events.

User Interface Toolbox Architecture

The user interface toolbox should provide:

- An application interface, which is the same for all applications. For example, this can be a syntactic server.
- An image processor generator, which generates an application-specific image processor. For example, the toolbox can provide a pretty-printing language that is used to describe the mapping of application data into a concrete representation. From a pretty-printing description, the toolbox generates the necessary connections between the application interface (e.g., a syntactic server) and an image processor (e.g., a formatter).
- A graphical user interface toolkit, that provides support for defining the external and internal interfaces. For example, this could be the Motif toolkit combined with a graphic library in the style of the graphic class of Interviews.
- A dialog manager toolbox, which defines and implements the application dialog manager. We describe such a toolbox below.

4.3 A Dialog Manager Toolbox

For the control integration between the application and its generated user interface, we propose to use our encapsulation technique. The application is associated with a node type corresponding to the application interface: inputs for entry points; outputs for notification points. The generated user interface is also associated with a node type, which is application independent. Finally, the dialog manager is associated with a node type that provides all the necessary inputs and outputs to connect the two.

The application can be coupled to its user interface by creating a network with an application node, a user interface node, and a dialog manager node. While the first two are atomic, the third is itself a network of nodes. The dialog

network is created using dialog fragments provided by the dialog manager toolbox.

Here, we only illustrate the principles of using elements of a dialog manager toolbox to create a specific application dialog. Suppose we want to create a `read` dialog, which one uses when the application needs to read a user specified file. From a purely functional view point, the read dialog returns the name of the file one want to be read by the application.

In terms of encapsulation, we can define a read node type with only one output port, i.e., *read*, which carries a string. Clearly, we have to associate the read dialog to some user interface widget, typically a button. For this we can use a predefined button dialog: it is an atomic node, of type **`button`**, with only one output, *activate*. Let's assume that we include such a button node in the read network. Next, we can choose another element in the dialog toolbox, that is activated by our button node. A natural candidate is a file selector dialog: it is an atomic node, of type **`file-selector`**, with an input, *activate*, and an output, *ok*, which carries a string. We include this in the read network after renaming the *ok* output to *read*. The final **`read`** dialog is shown in figure 5.

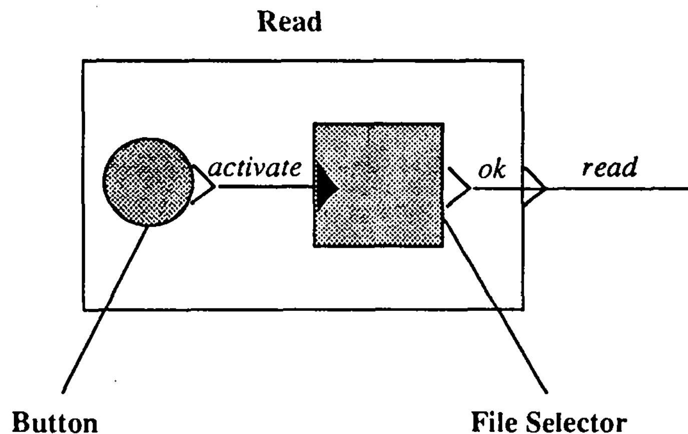

Figure 5: Read Dialog

It is the responsibility of the user interface toolbox system to associate the appropriate widgets to dialog fragments. In our example, a button widget to the button node and a file selector widget to the file-selector node. This is done by associating nodes to objects as described in the event handler section.

How does the read dialog behave? Initially, there is only a "read" button on the screen. The user depresses and releases the mouse within the button widget, this causes the user interface to send an abstract *activate* event to the event handler (the activate event contains the widget-id). The event handler

finds the button node associated with the widget-id and the *activate* signal is emitted within the `read` dialog network. On reaching the `file-selector` node this makes a file selector widget appear. The read dialog is emits a *read* signal, as soon as the file selector emits its *ok* signal. This happens on some user action, e.g., selection of an "OK" button provided by the file selector widget. (Of course, the user also has to have selected some file name!)

Interestingly, if the file selector remains visible on the screen then it is possible for the user to read another file without using the read button; he only uses the file selector. Here it is important that the communication between the dialog manager and the user interface to be asynchronous.

5 Conclusion and Related Work

The architecture presented originates from current work on software development environments and from work in developing an environment generator [4]. Our main concerns are: i) to make the techniques used in programming environments available outside the environment; ii) to make integration of technological changes as easy as possible. Note that the latter is a fulltime concern in our project [5].

This leads to a very open architecture where any component of the syntax, editing, and semantics classes can be replaced or modified. Also, many components of the proposed architecture can be used separately. For example, the formatter can be used alone by a tool, providing that this tool is interfaced with the document description language of the formatter. Of course, a natural way to generate this interface is with a specification of how the tool data can be transformed into the formatter's image structure. One possible implementation of such a translator is to have the tool data transformed into a tree of the syntactic server. Another possibility would be to use an approach similar to the Arcadia Specification Level Interoperability [30].

We believe the use of a distributed approach for programming environments is new, in particular the notion of syntactic server. Here, we claim that occurrence is the key concept. The notion of a formatter is already present in many systems, but the coupling of a formatter with both a syntactic server and an independent editor is novel. Our approach is to trigger the editor with structured oriented commands rather than the opposite. The notion of encapsulation comes from B. Fromme [14], revisited with concepts of concurrent programming. This makes our control integration mechanism independent of the implementation language of the tools. Finally, the user interface toolbox is a particular application of the concepts presented in this paper together with results on user interface builders, [18], [19], [8].

The proposed architecture is currently used as the basis for the implementation of the next version of the Centaur system. There already exists prototypes of the formatter, the encapsulation mechanism, and the user interface toolbox. A syntactic server is under development. For the coupling with external editors, we will experiment with Epoch [21]. Finally, we want to take advantage of a distributed approach to implement the user interface of interactive external components, e.g., language interpreters, theorem provers.

Acknowledgments

This paper would not have been possible without the work of several teams involved in the Esprit GIPE 2 project: the Croap team led by Gilles Kahn at INRIA, Sophia Antipolis; the Koala team led by F. Leygues at Bull, Sophia Antipolis; the Sema-Group team led by the author, in Sophia Antipolis. He especially wants to acknowledge D. Austry who is working on a syntactic server, L. Hascoët for his work on an incremental formatter and on occurrences, D. Dardailler for his work on graphic display components, L. Théry, V. Prunet, and A. Atie for their useful remarks on the encapsulation mechanism. Special thanks go to J. Incerpi for the numerous discussions we had and for her help in proofreading this paper.

References

[1] Andrew J. Palay et al., "The Andrew Toolkit: An overview", *Proceedings of the 1988 Winter USENIX Technical Conference*, Dallas, Texas, February 1988.

[2] B. Backlund, O. Hagsand, and B. Pehrson, "Generation of Interactive Graphic Design Environments", *9th IFIP Symposium on Protocol Specification, Testing, and Verification*, Twente, June 1989.

[3] G. Bolcer et al., "Chiron 1: Concept and Design", *Arcadia Document UCI-89-12*, October, 1989.

[4] P. Borras, D. Clément, T. Despeyroux, J. Incerpi, J. Kahn, B. Lang, and V. Pascual, "Centaur: the system", *Proc. of SIGSOFT'88, Third Annual Symposium on Software Development Environments*, Boston, USA, 1988.

[5] D. Clément et al., "Technical Annex of the GIPE 2 ESPRIT Project", SEMA-GROUP, Paris, France, 1989.

[6] G. Boudier, F. Gallo, R. Monot, I. Thomas, "An Overview of PCTE and PCTE+", *Proceedings of the ACM Software Engineering Symposium on Practical Software Development Environments*, SIGSOFT Software Engineering Notes, V.13 No.5, November 1988.

[7] M. Cagan, "HP Soft Bench: An Architecture for a New Generation of Software Tools", *SoftBench Technical Note Series, SESD-89-24 Revision: 1.4*, Hewlett-Packard Company, Software Engineering Systems Division, November 1989.

[8] L. Cardelli, "Building User Interfaces by Direct Manipulation", *Technical Report 22*, Digital Equipement Corp. Systems Research Center, October 1987.

[9] N. Carriero, D. Gelernter, and J. Leichter, "Distributed data structures in linda", *Proceedings ACM Symposium on Principles of Programming Languages*, Jan 1986.

[10] M. Devin, et al., "Aida: environnement de développement d'applications", *ILOG*, France, 1987.

[11] M. van Dijk and J. Koorn, "Implementation of a generic syntax-directed editor", *4th Review Report Esprit Project no 348*, 1989.

[12] Frame Technology Corporation, Inc., "FrameMaker Demonstration Document" 1988.

[13] P. Franchi-Zannettacci, "Attribute Specifications for Graphical Interface Generation", *Proceedings of the IFIP 11th World Computer Congress*, San Francisco, USA, 1989.

[14] B. Fromme, "HP Encapsulator: Bridging the Generation Gap", *SoftBench Technical Note Series, SESD-89-26 Revision: 1.4*, Hewlett-Packard Company, Software Engineering Systems Division, November 1989.

[15] J. Gettys, R. Newman, R.S. Scheifler, "Xlib - C Language X Interface, Protocol Version 11", *MIT project Athena*, February 1987.

[16] E. Golin, R. Rubin, and J. Walker II, "The Visual Programmers Workbench", *Proceedings of the IFIP 11th World Computer Congress*, San Francisco, USA, 1989.

[17] G. Huet, "Formal Structures for Computation and Deduction," *Course Notes*, Computer Science Departement of Carnegie-Mellon University, 1986.

[18] J.M. Hullot, "SOS Interface: un générateur d'interfaces Homme-Machine", *Actes des journées AFCET sur les Langages Orientés Objets*, BIGRE+GLOBULE, Janvier 1986.

[19] ILOG, Inc., "MASAI: L'outil de développement interactif d'interfaces graphiques", Paris, France, 1989.

[20] R. Ison, "An Experimental Ada Programming Support Environment in the HP CASEdge Integration Framework", *International Workshop on Environments*, Chinon, France, September 1989.

[21] S. Kaplan, "Epoch User Manual", *Technical Report*, University of Illinois, Urbana-Champagne,1990.

[22] M. Linton, P. Calder, and J. Vlissides, "Interviews: A C++ graphical interface toolkit", *Technical Report CSL-TR-88-358*, Stanford University, July 1988.

[23] R. Milner, "A calculus of communicating systems", *Lectures Notes in Computer Science*, Springer-Verlag, n. 92, 1980.

[24] E. Morcos-Chounet and A. Conchon, "PPML a general formalism to specify pretty-printing", *Proceedings of the IFIP Congres Bublin*, Springer-Verlag, North Holland, 1986.

[25] Open Software Foundation, Inc., *OSF/Motif Programmer's Guide, Revision 1.0*, 1989.

[26] S. Reiss, "Integration Mechanisms in the FIELD Environment", *Technical Report No. CS-88-18*, Computer Science Department Brown University, Providence, Rhode Island, October, 1988.

[27] Software Engineering Institute, "SERPENT Overview", *Technical Report CMU/SEI-89-UG-2*, Carnegie Mellon University, August 1989.

[28] R. Taylor, F. Belz, L. Clarke, L. Osterweil, R. Selby, J. Wileden, A. Wolf, and M. Young, "Foundations for the Arcadia Environment Architecture", *Proceedings of ACM SIGSOFT'88: Third Symposium on Software Development Environments*, Nov. 1988.

[29] T. Teitelbaum and T. Reps, "The Cornell Program Synthesizer: a syntax-directed Programm ing Environment", *Communications of the ACM*, vol. 24 (9), September 1981

[30] J. Wileden, A. Wolf, W. Rosenblatt, and P. Tarr. "Specification Level Interoperability", *Proceedings of ICSE'12*, Nice, France, March 1990.

Imprimé en France
par
. l'Institut National de Recherche en Informatique et en Automatique .

ISSN 0249-6399