User-Enhanceability for Organisational Information Systems through Visual Programming

Nikolay Mehandjiev[1] and Leonardo Bottaci[2]

[1] Dept. of Management Systems and Sciences
University of Hull
HULL HU6 7RX, England.
[2] Dept. of Computer Science
University of Hull
HULL HU6 7RX, England.

Abstract. Organisations that adapt rapidly require flexible software systems. Conventional system development methods are too slow for these organisations. One way to alleviate these problems is to empower members of the organisation, domain experts, to directly control and modify such systems (user enhanceability). This paper considers the applicability of visual languages as an enabling tool for user enhanceability. Previous systems in this area have succeeded only for narrow application domains and have failed to scale up. This paper highlights some major problems in such an endeavour, presents a generic architecture that addresses these problems and discusses a user enhanceable system for workflow applications that was implemented using this architecture.

1 Introduction

1.1 Why We Need User Enhanceability

Information systems for the support of organisational group activities (e.g. office and workflow systems) are notoriously difficult to build and maintain due to the speed of organisational changes and the complex social aspects of group work and organisational processes. These systems must support higher-level organisational coordination activities such as activity management, task scheduling and workload distribution, but these high-level activities are quite vulnerable to organisational change and their social context makes them inherently difficult to capture and formalise. As a result, frequent modifications are needed to either reflect organisational changes (often known as adaptive maintenance) or to better match the system to the organisational and user needs (part of perfective maintenance).

The large costs and time-scales involved in modifying software are often attributed to the necessity of translating user requirements to code modifications. This translation is generally performed by software professionals and, as Poo and Layzell state in [27], the process is 'often fraught with misunderstanding and poor translation, which result in unwanted changes, and so the process

repeats, adding to the total maintenance costs'. The general reasons for miscommunication between software professionals and end users are summarised by Novick in [25].

Software professionals play a vital role during development by ensuring that the system is based on a well structured architecture. However, during maintenance their role (assuming that they have indeed produced a good design) is less strategic and for most modifications they are acting simply as translators or programmers. Some researchers and practitioners believe that domain experts would be more effective than software professionals in performing the majority of the needed enhancements [18, 22, 1]. Enabling these domain experts, which we call *application managers*, to perform a large subset of the information systems enhancements with little or no help from software professionals is known as *user enhanceability*. Other names for user enhanceable systems are 'radically tailorable systems' [23] and 'configurable systems' [7]. Although we assume that application managers are computer literate, they are not expected to know a conventional programming language, nor be familiar with programming techniques and algorithms. Apart from helping to keep information systems up-to-date, user enhanceability [16] will also give application managers a powerful instrument to explore different processes and to introduce organisational transitions in a gradual way.

1.2 Our Approach to Achieving User Enhanceability through Visual Programming

Visual programming languages describe a program through graphical or spatial expressions rather than through character sequences as in conventional programming languages. They are generally considered to have potential for user-level programming [30, 24], but if a visual language is to be suitable for application managers then it should not be simply a programming language which has a diagrammatic as opposed to a textual representation. Such a language solves none of the problems of programming except perhaps difficulties with syntax. Instead, a visual language must render irrelevant the low-level problem solving activities of conventional programming [12, 28, 30, 29, 19].

The visual language must exploit, as far as possible, the fact that these application managers will not be tackling general purpose programs but will be solving problems that are specific to their particular domain, for example constructing a depreciation formula in a spreadsheet, or describing the route of an electronic form throughout an office. The visual language should thus provide task-specific programming constructs, i.e. constructs that represent concepts from the application domain. The graphical dimension of the visual programming language can be exploited to present these concepts effectively. Such a visual language is often called a task-specific or domain-oriented visual programming language. An extended argument for the benefits of task-specific programming is contained in Nardi's book [24].

Unfortunately, a number of problems have prevented the successful application of domain-oriented visual programming in complex domains [8, 5], and they

have so far been successful only for comparatively simple target domains, for example spreadsheets [23] for accounting and statistics, form painters for database access, LabView for signal processing, Khoros for image processing. The failure of visual languages to scale up has been recognised and some problems of the general-purpose visual languages have been investigated in a recent article by Burnett *et al* [5]. However, the problems faced by domain-oriented languages are rather different than the problems of general-purpose ones.

In the remainder of this paper we discuss the problems of applying domain-oriented visual languages for user enhanceability of organisational information systems and then we describe a novel architecture that addresses these problems and achieves enhanceability at two levels:

- User enhanceability, where application managers modify application functionality to reflect changing organisational requirements
- Meta-enhanceability, where software professionals modify enhanceability facilities to reflect changes in the enhanceability requirements.

The paper concludes with a brief report about experiences from ongoing tests of ECHOES, and with a review of the relevant research work, where the novel aspect of the proposal are identified.

2 Applying Visual Languages to Complex Information Systems

2.1 Management of Complexity

One of the first problems we face when applying visual languages to organisational information systems is one of complexity. We need to find a compromise between hiding complexity from the application manager and providing an expressive visual language. Four strategies for complexity management are appropriate in this context:

Multiple Descriptions Since a well designed domain-oriented visual description should employ only a small number of high-level abstractions [24], complex concepts are often best presented by using multiple descriptions, For example, organisational structure should have a different description from that used for information processing and different yet again from the structure of the information. Also, by having a choice of different enhanceability formalisms and facilities, there is no need to treat all organisational processes in the same way.

These multiple descriptions must of course be integrated to help application managers to comprehend and navigate between them. The descriptions thus form a visual language, where each description communicates to the application manager a small coherent aspect of the application.

Assumptions The descriptions employed by domain experts are often informal and incomplete in the sense that a lot of detail is not articulated but taken for granted. This is inevitable and indeed a desirable consequence of abstraction. For example, the activity of sending a form to a specific office worker could be easily presented in a visual language as the dragging of a form icon to an office worker icon. However, this representation hides many details. For example, office workers receive many forms and so they must be queued in some way. What is the queue discipline? One assumption may be that all queues are served in first-in first-out (FIFO) manner.

Often, as in fourth-generation languages (both visual and textual), the software designers bury these assumptions deep in the code, hoping that they match user's expectancies and "common sense". Unfortunately, when an application manager tries to comprehend and modify the system, his or her expectations may be quite different from the assumptions made by the designers. For example, the application manager may rely on priority-based queue processing, and would therefore expect a form or letter, marked "Urgent" to be processed immediately.

We argue that such assumptions are a part of the application domain and therefore they should be explicit. They should be available on request to the application manager. Hopefully, the application manager will not need to modify assumptions, but if this proves necessary, assumptions can be replaced by visual descriptions, thus enhancing the visual language. Assumptions have well defined interfaces to facilitate replacement by a visual description.

Exploratory Enhancements In any complex system it is difficult to predict all the consequences of a specific enhancement. Therefore, in an ideal user enhanceable system it should be possible to perform modifications in an exploratory style. That means that the boundaries between modification and execution must be removed and rich animation and feedback techniques provided. The application manager should directly manipulate the application system through its domain-level representations, thus fully exploiting user-level metaphors for domain concepts. Visual languages that provide this style of working are referred to as *responsive* visual programming languages [5], or as visual languages with a high degree of *liveness* [33]. In these languages there is no separate compilation step, instead the user actions are immediately executed and their effects shown. Examples of such visual languages are spreadsheets, Forms/3 [4], OVAL [23] and HI-VISUAL [14].

Modifications Support Whenever possible, important consequences and side-effects of a change should be pointed out to the application manager, even if they do not belong to the visual description where the change is performed. For example, when an office worker is deleted from the organisational description, the description that shows flow of work between activities should also be updated to exclude the deleted worker. This update should then be brought to the application manager's attention, since the deletion may affect other activities.

2.2 Changing Enhanceability Requirements (Meta-Enhanceability)

Enhanceability requirements cannot be guaranteed to be stable and there is sometimes a necessity for software designers to change the manner in which the application is modified (meta-enhanceability). Indeed, very high level, application specific languages are more vulnerable to such changes since they typically lack the universality of conventional programming languages.

A substantial meta-enhancement would be to substitute one visual description with an alternative. For example, when describing the personnel of an organisation, we may substitute a taxonomy that is based on specialising job descriptions, to an organisational chart, which shows the authority structure. The most radical enhancement would be to develop a new visual description to control some previously unvisualised aspect of the domain, for example a visual description to interact with a queue of forms. Providing facilities for such meta-enhancements while preserving the underlying application is one of the problems addressed by our architecture.

3 An Instantiation of the Proposed Architecture

A crude description of the proposed architecture of pluggable visual descriptions is presented in Figure 1. It consist of a number of visual descriptions and assumption modules, which interface to an application system. However, the best way to present the architecture is to understand the motivation behind it. This motivation is illustrated below in an extended use scenario within ECHOES *(Easily CHangeable OfficE Systems)*. ECHOES is a prototype system, implemented in Smalltalk-80 with VisualWorks and HotDraw — a generic drawing editor described in [17]. ECHOES instantiates the architecture to provide a user enhanceable workflow system that may be used for order and invoice processing, insurance application approval, etc.

3.1 The Visual Language

The domain is abstracted into the following user-level visual descriptions: information flow, service definition, organisational structure, information taxonomy and information appearance description.

Information Flow Visual Description ECHOES is a general purpose architecture for end-user development systems but to explain ECHOES we will use an example workflow system. This kind of system is likely to be familiar to readers and the particular system we describe is part of the processing of university undergraduate applications within a particular department. As shown on Figure 2, the application forms are initially checked for completeness, then some of the applicants are selected for interviewing and others are rejected. After the interview an offer is decided for the particular applicant, the case is filed and

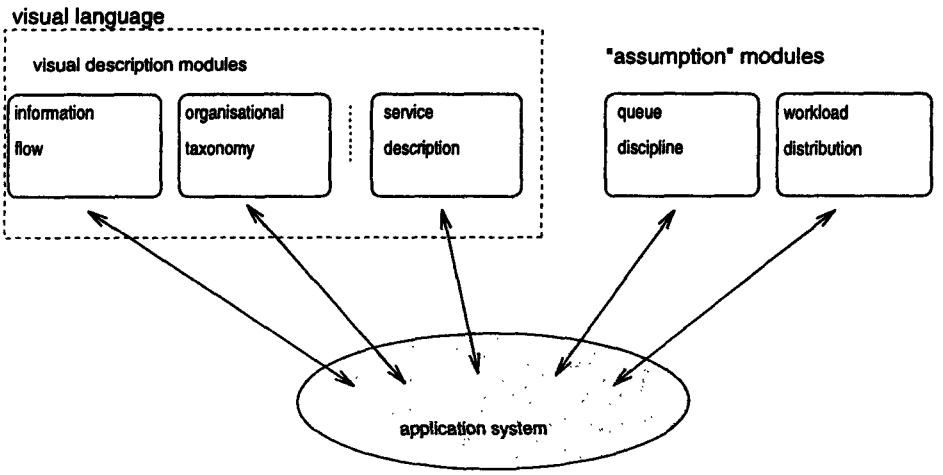

Fig. 1. A crude description of the architecture behind ECHOES

letters are sent as appropriate. The results are sent to the university admissions system for processing.

The flow of information between office clerks is represented as an information flow diagram. It is based on the data flow paradigm and, as shown on Figure 2, the diagram shows the flow of information and control messages between the services that constitute the application system. A service is an information processing activity performed by a participant in the workflow, a clerk.

A service is represented as an unobscured rounded rectangle, all the clerks that perform the particular service are represented as rounded rectangles that are stacked behind it. The message flows are represented as arrowed lines that connect the source and the target of the message. To reroute messages the application manager may directly move the message arrows to new source or target services.

If the application manager, for example, needs to add a new service to the system, he or she selects the appropriate service symbol from the palette on the right and places it on the diagram as shown with the dashed arrow. The dashed lines have been superimposed on the screenshot to show the interaction dynamics. The visual description will convert the application manager's action into a sequence of "low-level" commands that add a new service to the application system.

When the application manager adds a new service to the information flow diagram, he or she should also specify who may perform the service. The application system knows that each service is performed by one or more office clerks, and therefore detects the need for additional information. The application broadcasts a request for this information to all modules currently plugged to

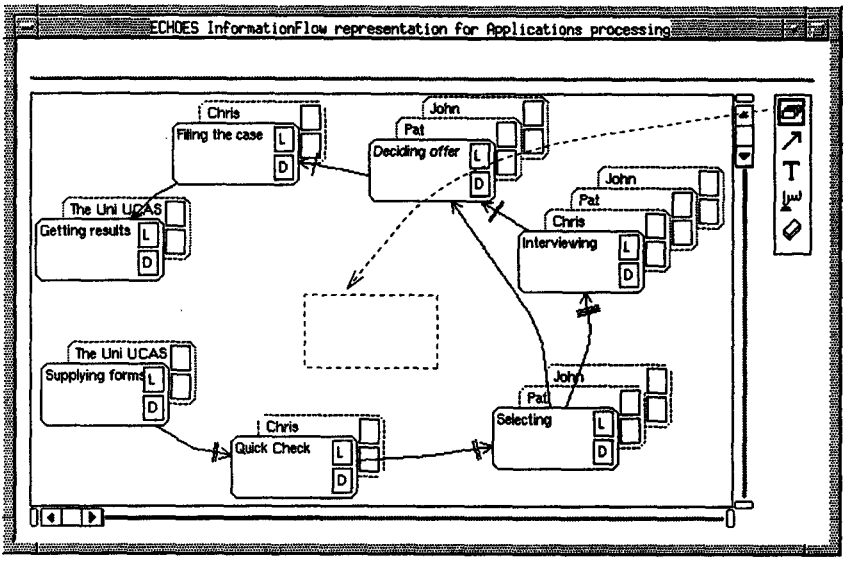

Fig. 2. Example information flow visual description for university admissions

the application system, regardless of whether they are visual description modules or "assumption" modules.

Organisational Visual Description In ECHOES, the assignment of services to office clerks is performed via the organisational description. It groups services into "job descriptions" and then classifies clerks in a prototype inheritance taxonomy, based on their job descriptions. An example is shown on Figure 3 ('Academic' is the job description, 'John' and 'Pat' are the office clerks that have this job description).

Continuing the example from the previous section, in response to the broadcast request for which office clerks are to perform the new service, the organisational description module responds to supply this information. It opens the window on Figure 3, which shows the organisational taxonomy, and asks the application manager to select either a group of people, linked by a common job description such as "Employee" or an individual office clerk such as "Pat". If, as shown on the Figure, the application manager selects 'Pat', the description module will ask him or her to confirm whether the new service will be added only to Pat's job description or to the job descriptions of all "Academic" staff. Confirmation is necessary since, if the answer is 'just Pat', a new job description will be specialised for her (as shown on Figure 5).

Now the system has all the necessary information to create a new service as a part of the application. After the service is created, the application system will notify all "plugged in" modules. In response, some of the visual description

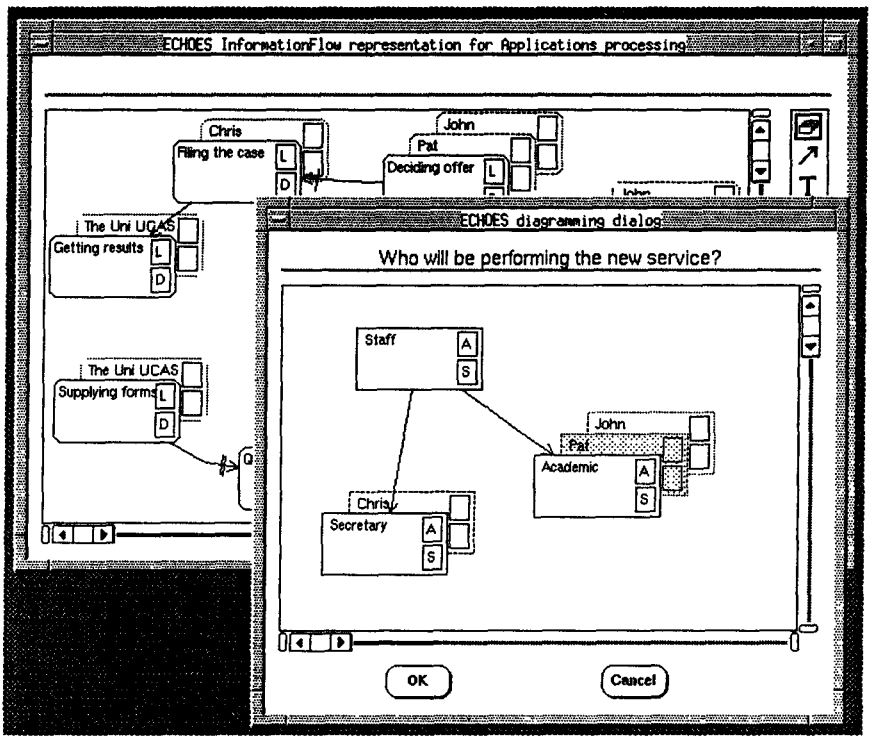

Fig. 3. Organisational description module responds

modules will update their representation of the application system. For example, the information flow module will place the new service on the information flow diagram as shown on Figure 4, and stack Pat's oval behind the service's symbol, since it has read the current configuration from the application system.

The new service will initially appear with a default name and priority. The application manager can rename it and change its priority by clicking on the "S" button of the relevant job description as shown on Figure 5.

The application manager may then proceed to connect the new service to other services with message flows. When a new flow is created on the diagram, the system needs to know the type of information on this flow and this request is answered by the information descriptions, the subject of the next section.

Information Descriptions In ECHOES, information is modelled in terms of forms. On the information taxonomy visual description, the type of each message flow is specified in a hierarchy which defines specialisation and composition relationships between these forms. Manipulating this taxonomy is performed in a similar manner to manipulating job descriptions on the organisational taxonomy.

The structure and the appearance of each form is controlled by the applica-

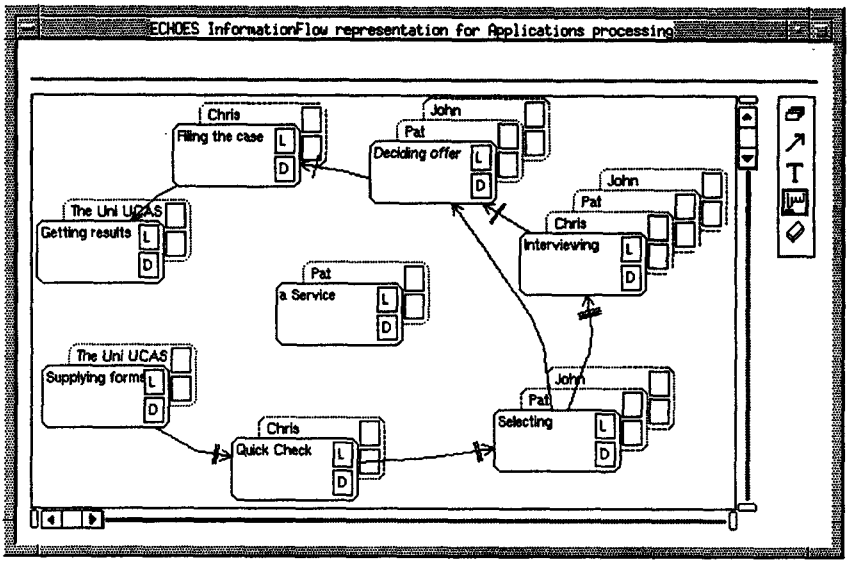

Fig. 4. The new service appears on the information flow description

tion manager directly manipulating the form's elements (*i.e.* input fields, menus, formulae) as in a standard form painter. Indeed, the standard form painter supplied by the implementation platform VisualWorks was used in ECHOES, since form painters are not novel.

Service Visual Description When the new service "Selecting Disabled Applicants" is created in our example, a default process description is associated with it. In its simplest form, the service will simply present forms that arrive on incoming flows to the office worker. The worker can process the form and then press the 'OK' button, then the service will send the result to an output flow. This default service description is shown on Figure 6. Four dashed arrows have been added to the screenshot to show the dynamics of the interaction.

The visual description of the processing that takes place in services, integrates the event-driven and the procedural elements of a service in a single visual description. As shown in Figure 6, a template-like frame surrounds the description. Events that control the processing are listed on the top of the frame. The input flows are listed on the left of the frame and given names, the output flows are listed on the right. Interaction with users is also modelled in terms of form-based messages exchanged between the service and the office worker, but in this case the form is displayed on the screen to both present output and gather input from the office worker. These interaction flows are designated with a computer terminal icon and do not appear on the information flow diagram.

The service description has highlighted the second output flow with a ques-

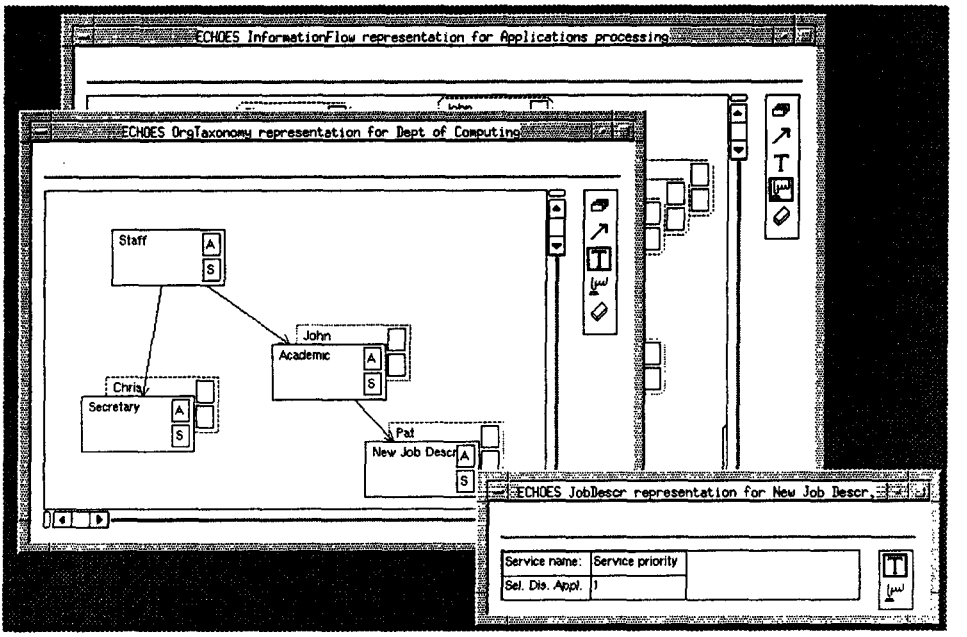

Fig. 5. The new job description has been created

tion mark, because it is not yet assigned. This prompts the application manager to modify the description to take this flow into account. Indeed, if the default service processing is not satisfactory, the application mānager has to modify the service processing description. A description could be copied from another service and then modified to reflect the new functionality required.

Processing control flow is animated. The current state of the processing is represented as a token moving through a network of states. The triggering events are shown to control the opening and closing of switches that in turn control the movement of the token by enabling and disabling certain transitions. On its way through the network the token passes through the so-called action blocks, where it triggers a simple sequence of actions by stepping through them. The sequence of actions in each action block is represented in a separate window, as shown on Figure 6.

3.2 Assumption Modules

After the newly created service 'Selecting disabled applicants' is connected with information flows to the rest of the diagram, forms will start queueing on the input flow of the service. When Pat comes to work and opens her 'To Do' list, the new service will appear. If Pat activates the new service, the service object within the application system will try to get the first item on the incoming queue

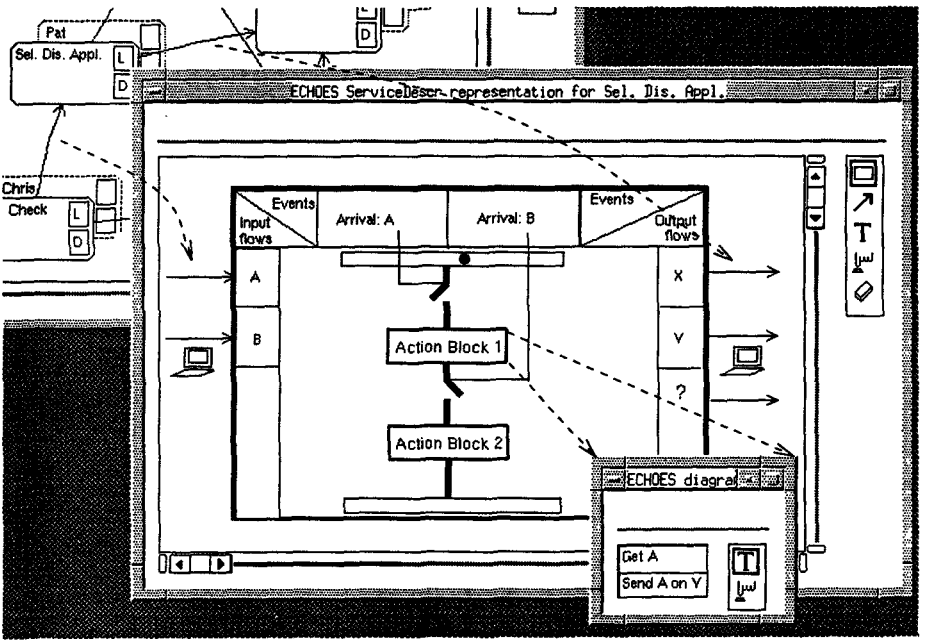

Fig. 6. The default visual description of the new service

from a special coordination object. This coordination object is shared between all the clerks that perform a given service. It distributes their workload and manages their incoming queues. This coordination object contains the scheduling algorithm for retrieving the next item on the queue for a particular clerk.

Currently there is no visual description in ECHOES, whereby the application manager can specify this algorithm. In a traditional fourth generation language this algorithm will have been buried in the implementation code of the coordination object. In ECHOES, however, the specification of this algorithm is encapsulated into an assumption module, which is integrated into the system in an identical manner to all other visual description modules.

When the new service was created by the application manager, the corresponding coordination object was also created, and the specification for its queue discipline was defined by the assumption module. The current specification in the assumption is that all queues are served in 'first-in first-out' fashion.

So, like visual description modules, assumption modules supply application-specific information. However, unlike the visual description modules, the assumption modules obtain the necessary information by performing a pre-specified algorithm that can not be visually manipulated by the application manager.

However, in our scenario, the application manager may wonder why a certain

form that was marked 'Urgent' has not been processed immediately by Pat, but has been waiting on the queue together with all other forms. In this case the application manager may try to open the queue in the way he or she would open any other element on the diagram that has a visual description (e.g. a service symbol expanded into a service definition diagram). The architecture does not distinguish between visual descriptions and assumptions but simply propagates an "open description" request for the relevant application aspect. Since our queue assumption module does not have a visual description, it will instead open up a text window containing a explanation of the assumption that is coded in the algorithm of this module. In our case this would read "All information items on a queue are processed in a first-come first-serve basis".

So, although the application manager cannot, at this level, control the assumption contained in the corresponding assumption module, he or she can at least be made aware of it.

4 Pluggable Visual Descriptions Architecture

In this section we will describe the architecture of pluggable visual descriptions, which underlies ECHOES. This architecture allows an application system to be controlled through multiple visual descriptions. The use of multiple descriptions of a single application system is similar in concept to the model-view-controller paradigm used in Smalltalk and described in [11]. An outline of the proposed architecture of "pluggable visual descriptions" is presented on Figure 7.

In this architecture the domain model encodes the facts that hold all applications in the target domain of organisational systems. This model provides a language for controlling the application system.

A higher level domain-oriented visual language, consisting of several visual descriptions, is used to construct and modify application systems. Each description presents a coherent aspect of the application to the application manager in high-level task-specific concepts. The assumptions made when creating the visual language are made explicit in "assumption" modules.

4.1 Visual Description Modules

Each visual description module is composed of two parts as shown on Figure 8. The graphical interface part enables the application manager to directly manipulate diagrammatic, table-based or some other graphical representation of a part of the system. The second part is the translator, which links graphical actions and graphical entities to operations and entities in the application. In one direction, it transforms the application managers' manipulations of diagram symbols into operations performed over application-level objects. In the other direction, it maps the application objects to concepts and symbols on the diagram. The translator ignores the irrelevant and cosmetic user gestures such as moving a node in a graph-based diagram.

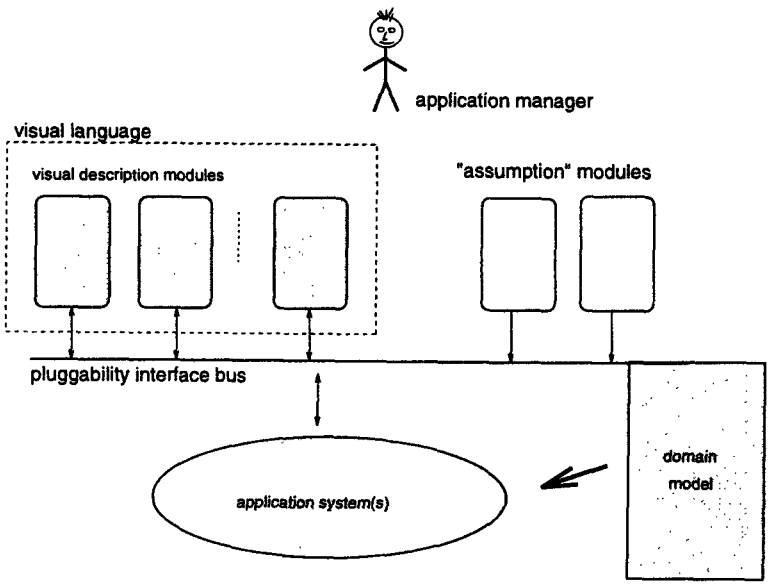

Fig. 7. Generic architecture for user enhanceability — "pluggable visual descriptions"

For example, when a new symbol representing a service was added by the application manager to the diagram in the scenario of Section 3, the graphical interface passed the event on to the translator, which translated the event as a request to the application to generate all corresponding application object(s) that implement this service. After the application objects were generated, all translators were notified that the application has changed. The information flow description then translated the new application objects to a high-level representation of a service, which was displayed by the graphical interface part. Some other visual descriptions also added the new service, the rest ignored the notification.

Similar mechanisms of interaction between the application system and a translator are also used in the following two cases:

prompting for information needed to complete a change Often, as in the case of adding a new service above, the application manager has to supply additional information. The application will then broadcast a request for this additional information to all modules. Then the translator of the first relevant visual description module will arrange for the description to prompt the user for this information and will supply it to the application. The other relevant modules will not be activated.

implementing hypertext-style links To support modifications, concepts from different visual descriptions are linked with hypertext-style links. These links

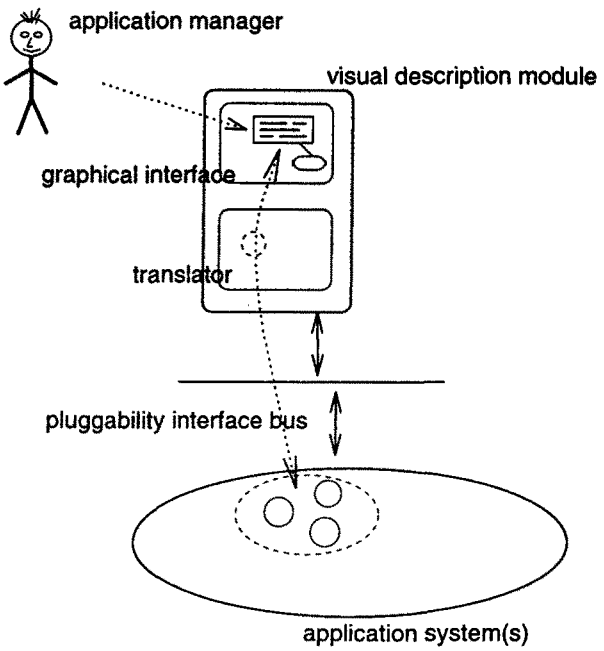

Fig. 8. Structure of a visual description module

serve as navigation paths and as change propagation links. For example, when a high-level concept has been modified, or even just selected, in one visual description, the application manager should be demonstrated the effects on all relevant concepts from other visual descriptions. This is implemented by the following mechanism. When application manager selects a high-level concept, its translator will "flag" the corresponding application objects. This event will be propagated to all other translators, and some of them will in turn highlight the high-level concepts that are linked to some of the "flagged" application objects.

For a given domain model, and, indeed, even for a given application, various descriptions may be added or removed (i.e. "plugged" in and out of the system), thus catering for cases when the requirements for enhanceability change. i.e. for "meta-enhanceability". An example of a meta-enhanceability scenario is given later.

The key feature that enables such plugability is that a visual description does not contain any information that is necessary for running the application, such as the state of the application objects, their functionality or the relations between them. A visual description only keeps representational information such as the position of a symbol in the diagram. Indeed, a completed application system should still run even after all visual descriptions are detached from it.

4.2 Assumption Modules

Assumption modules implement aspects of the domain that need to be available for inspection, but hopefully should not require modification. Thus an assumption module does not need a visual representation, but it should be narrated to the application manager on request, possibly by devising a "help"-style textual description. If later, as a result of accumulated experience in using the software, an assumption turns out to be inappropriate, it can be modified by software professionals. If the application manager wants to control the assumed aspect, then it can be replaced by a visual description. This will expand the visual language.

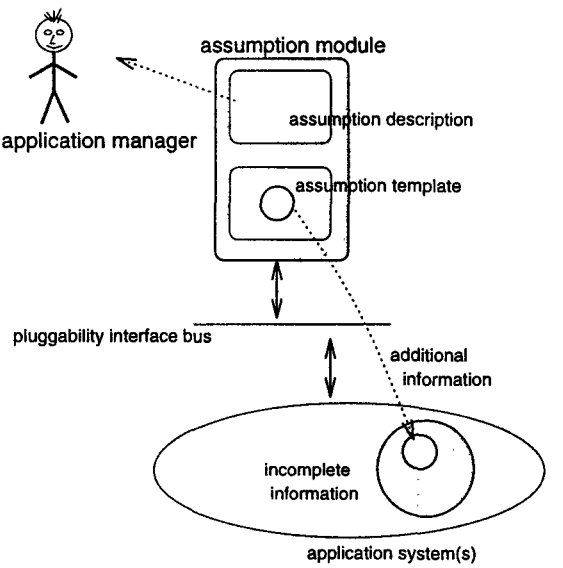

Fig. 9. The structure of an assumption module

Handling these requirements is best performed by separating the assumption in a separate assumption module as shown on Figure 9. In our extended ECHOES example we have encapsulated in such a manner the template algorithm dealing with the discipline of message queues. An assumption module also contains the text describing the algorithm, and behaves in a similar way to the visual description modules:

- It supplies the design assumption contained in its software template to the application. This assumption will then be used when formally interpreting the informal concepts and commands within the visual descriptions
- It displays the textual description of this assumption to the application manager on request.

Since assumption descriptions are explicit and available for inspection by the application manager, we overcome the problems with the fourth generation languages, where, for example, the design assumption about how users would do a search through a data table is hidden in the code of the language interpretor. All the user of such a system would see is the FIND command, which may not operate in the desired manner.

In essence, the assumption description modules complement the current set of visual descriptions and serve as place-holders for future visual descriptions to be added to the visual language. These novel assumption descriptions are a major factor that enables our user enhanceability approach.

4.3 Meta-Enhanceability

Meta-enhanceability is the facility to change the means and ways in which a system can be enhanced. In this section we will extend the scenario from Section 3 in order to demonstrate how building ECHOES using the "pluggable visual descriptions" architecture extends what is enhanceable by adding a visual description that controls the queue discipline instead of the current assumption module. Similarly, one type of visual description could be substituted for another by unplugging the relevant visual description module and plugging a new one in its place.

For this scenario, let us suppose that the queue discipline for all queues in the system is 'FIFO', and that this is unsatisfactory. The discipline is implemented as an assumption, but a more flexible arrangement is required, where the application manager can control the queue discipline. We decide to develop a new visual description to allow the application manager to see the status of each queue and to manually re-arrange the order of the items on the queue as required.

This visual description module incorporates a modified version of the algorithm from the old assumption module. The information items on the queue are ordered first-in first-out within two subsections — urgent and non-urgent, the items from the urgent subsection are processed first. The visual descriptions shows the information items ordered in the two subsections of the queue, but also allows the application manager to manually override this ordering at will by directly moving items.

The old assumption module is then removed from the system and the new visual description module is plugged in its place. The new module also specifies the queue ordering algorithm as did the old one, but in addition it can issue requests for the coordination object to override this algorithm and to re-arrange the default order.

5 Discussion and Limitations of the Proposed Approach

5.1 Discussion of the Coherence Problem

To allow different visual descriptions to be easily plugged in and out of the application, which is essential for meta-enhanceability, we require that each visual

description should be directly connected only to the application and comparatively independent from the others for its operation. Also, the application should not be dependent on the visual descriptions for its execution. However, a problem exists with this architecture. One visual description may put the application system into a state that has not been anticipated by another visual description, and the latter then becomes invalid.

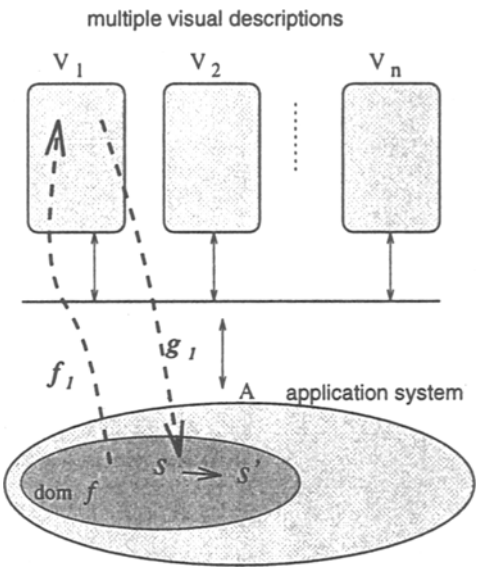

Fig. 10. The pluggable visual descriptions idea

In Figure 10, V_1, V_2 and V_n denote different visual descriptions, attached to the application system A. Creating and maintaining a particular description V_i could be presented as a partial view function f_i from the set of all application system states S_A to the set of corresponding visual description states S_{V_i}, so that for any $s_A \in S_A$ either there exists $s_{V_i} \in S_{V_i}$, such that:

$$s_{V_i} = f_i(s_A)$$

or:

$$f_i(s_A) \text{ is undefined.}$$

For example, such a view function will convert the state "snapshot" of a workflow application system to a description that shows the workload of every worker involved. Implementing the view function is the responsibility of the visual description, since the application system should not be dependent on any visual description.

If visual descriptions are to be used for modifying the application, our visual description V_i should also transform user operations within the visual description into commands requesting modifications of the application. In other words, the description should implement a control function g_i from a set of user operations O_{V_i} to a set of corresponding sequences of commands for modifying the application O_A. For example, if a user adds a new service to the information flow visual description, the visual description should generate a sequence of commands that generates a new service in the application system. There exists $o_A \in O_A$ for every $o_{V_i} \in O_{V_i}$, such that:

$$o_A = g(o_{V_i})$$

Since a single visual description deals with only one aspect of the domain, it is highly likely that any view function f_i is not defined for all application states:

$$\text{dom } f_i \subset S_A,$$

i.e. that exists s_A, such that $s_A \in S_A$, and $s_A \notin \text{dom } f_i$

This means that visual descriptions will be undefined for some application states. However, undefinedness is not the real problem since a visual description can alert the application manager by showing a special message instead of a diagram, but detecting undefinedness is. There are two alternative solutions to this potential problem. The first alternative is to ensure that the application state stays always within the definition domain of the visualisation function f_i, and indeed, considering the definition domains of all visual descriptions as shown on Figure 11, that the application state stays always within the intersection of the definition domains of all visualisation functions *i.e.* for all s_A, for which $s_A \in S_A$, we have

$$s_A \in \bigcap_{i=1}^{n} \text{dom } f_i$$

Enforcing such a solution, however, contradicts the "independence" of visual descriptions, a basic principle of the architecture, since, if the visual descriptions are indeed independent of each other, there are no means to ensure that this union in not an empty set, as illustrated on Figure 11(a).

Even if the visual descriptions have been designed to work together and therefore their definitional domains overlap as shown on Figure 11(b), keeping the application state within the area of overlap would again contradict to the principle that the application system should execute independently from the visual descriptions. Indeed, the transitions between application states are based on the application's internal logic. If the application is independent from its visual descriptions and the translator functions are defined within the visual descriptions, the application can not possibly detect whether a state transition will take it out of the definitional domain of a description, so it can not stop the transition from happening nor issue a warning. And since the application is the only one to approve changes of its state, the visual descriptions can not prevent these transitions either.

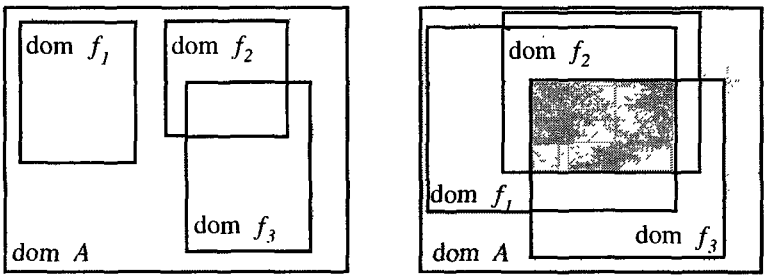

Fig. 11. Two cases of visual description definition domains

The second alternative is to not limit the state of the application system. In this alternative, each view function f_i should be constructed in such a way that it can detect when the application is in a state for which the function is not defined, *i.e.* $s_A \notin \text{dom } f_i$, by testing some pre-conditions. If the test fails, the visualisation function will issue a warning to the user instead of creating a visualisation.

Indeed, testing for pre-conditions is now encouraged as a good software implementation practice. Furthermore, in most practical cases the view function will perform this testing as a part of its algorithm, and therefore this solution will not bring about a big increase in the development efforts and costs.

This solution could be used to add functionality to the visualisations of the application. For example, different users may be allowed to see or modify only certain visual descriptions according to their authority. Also, the users of a workflow application system, for example, may want to see different visual descriptions in different business situations.

5.2 Other Issues

limits of meta-enhanceability Another technological issue to consider is the limits of meta-enhanceability for a given domain model. The variety of concepts and paradigms used within the visual language is constrained by the semantics of the underlying domain model, since each high-level concept or an operation from a visual description corresponds to a set of objects or operations from the domain model. Therefore the level of abstraction in this model should be carefully selected, if a large variety of user-level visual languages are to be mapped onto the model.

applicability to other domains The architectural framework presented in this article does not seem to contain domain-specific constructs, and several similar architectures as I-CASE, C-CASE[31] and MVC, are deemed applicable for professional software development in a variety of domains. However, our

practical experiences of applying the framework are narrow, since ECHOES targets only the domain of workflow application systems. Therefore, the architecture's applicability to other organisational and non-organisational domains can not be tested without substantial further investigations.

organisational factors Different organisational factors play an important role for the success or failure of user enhanceability. For example, the application manager should be someone with time to devote to maintaining the application, and with the power to implement the necessary changes in work patterns. Also, some of the participants in the system may see the application manager's role as holding too much power, and may obstruct the process of implementing changes. Another danger is that, when the application manager leaves the organisation, the system could be left in a poorly documented state, after a large number of modifications have caused design deterioration.

system design deterioration It is widely accepted that any software enhancements cause deterioration of the system design[3, 20]. This factor is particularly important in an user enhanceability scenario, where application managers lack training in software design principles. Hopefully, since application managers are domain experts, working with domain-oriented model of the application will help them to avoid changes that cause deterioration of this model and of the design structure of the application.

risks of end user development Allowing end users to create and modify applications software, can lead to a large number of errors if appropriate support is not provided. For example, Panko in [26] points that some consultants' personal experiences show that something like a third of all operational spreadsheet models contain errors. One consultant from Price Waterhouse reported 128 errors within four large spreadsheet models[6]. To alleviate these problems, application managers need to be supported not only with better tools, but also with better training and professional help.

6 Our Experiences so Far

We are still at early stages of testing ECHOES, so the experiences reported here should be treated as preliminary first results only.

Implementing ECHOES in Smalltalk did not require departures from the architectural framework outlined here. The Smalltalk dependency mechanism was used to implement the plugability bus. The number of visual descriptions turned out to be high, since attribute tables, action block descriptions and other auxiliary windows were implemented in a way identical to the major visual descriptions described in the paper.

We have used ECHOES to develop two workflow applications. The first is a simplified version of departmental system for processing student applicants, which was used in section 3 to present ECHOES. The second one is a workflow system to support production of a central university timetable. The first system was used as a case study at early stages of the project, and its requirements were

captured using a structured analysis method. They were recently converted into a workflow application, which was used to test ECHOES on several scenarios. The feedback was used to appropriately modify ECHOES. The requirements of the second application were captured using the ECHOES language as a requirements formalism, and this seemed to work well for this limited size project. When showing diagrams to the University timetabling administrator, we were surprised to find out the large extend to which her domain expertise did reduce the necessity for training in the ECHOES formalisms.

We have designed, but have not yet implemented, two pilot experiments, where ECHOES will be used by two domain experts to perform enhancements in the two applications developed. While the first experiment will involve a departmental administrator who has not taken part in the development of the university applicants processing application, the second one will involve the domain expert who was helping us to develop the timetabling system. On these experiments we hope to obtain feedback comments that will help us develop better appreciation of how well does the tool and the architecture support user enhancements.

We should stress here that we do not currently plan to conduct statistically reliable evaluation studies for three main reasons. Firstly, such studies will require a large number of participants to take into account diversity of programming backgrounds, aptitudes and abilities. Secondly, the software is still a "proof-of-concept" prototype and is not stable enough for usability evaluation. Finally, good evaluation of enhanceability will require long-term observation of enhanceability practices by real application managers.

7 Related Work

Multi-description Based Visual Languages Multi-description based visual languages are presented in [2] and [13]. Indeed, Visual Toolset [2] and MViews [13] do exploit the idea of handling domain complexity by using a multiple visual and text descriptions as a part of a multi-paradigm language. However, they are not oriented to a particular application domain and as a result they take different considerations into account. For example, Visual Toolset is oriented towards general programming. Therefore, while the elements of the specialised visual language in our paper map directly to the domain concepts and to the model structures, the users of Visual Toolset have to perform this mapping themselves for each particular application domain, which requires a higher level of programming expertise and familiarity with the programming language concepts such as data structures, functions and procedures.

MViews is a model for general-purpose software development environments, where the main concern is integrating the different views and the change propagation. It has evolved in the domain of Prolog programming, and therefore lacks these elements of our architecture that support organisational and office domain specifics such as the assumption modules.

Visual Languages for Organisational and Office Systems Some visual languages that address the domain of organisational and office systems are "Officeaid VPE"[10], HI-VISUAL[14] and Regatta VPL[32].

"Officeaid VPE" and HI-VISUAL have a restricted scope in the sense that they are used for the description of single office tasks only, without integrating them in an office processing framework. The first of them is based on a flowchart representation of office activities, partly built by programming-by-example techniques, the second one exploits the object semantics of overlapping iconic representations of office concepts.

Regatta VPL, on the contrary, targets the overall work process in an office or organisation. It allows end-user modification of process descriptions through the use of a graphical planning tool, thus evolving and improving the process description on-the-fly. It is a representative of a subclass of the workflow systems described in [15]. The systems from this subclass are used to visually describe organisational workflows. However, these systems tend to take a single, highly abstract view of the organisational processes, and are not designed to accommodate radical meta-enhancements.

Enhanceable Systems Oval[23] and of Cosmos[7] are concerned with the user enhanceability of office systems for cooperative work. Oval aims to allow users to modify their applications, but users are meant to develop their system from scratch. The system does not offer visual programming tools as such, nor support for coordination activities in its user-level language.

Cosmos, on the other hand, does provide higher-level coordination support, but the model is be configured by using two text-based languages - a scripting language and a communication structure definition language. The authors of this paper believe that making the system configurable by ordinary users could be achieved on a textual level with the help of good textual editors as opposed to developing diagrammatic versions of languages.

A third system, proposed by Oei *et. al.* in [9] handles not only enhanceability, but also meta-enhanceability. Starting from an empty system, the application evolves as a result of a stream of update requests that are processed and then validated. However, the provided enhanceability facilities are not targeted to the level of computer expertise that domain experts are expected to have.

8 Summary and Conclusions

If visual languages are to be effective in building user enhanceable information systems, then one must address the complexity of the domain, the specific requirements of user-level programming and meta-enhanceability.

This article proposes a novel conceptually integrated approach to these problems, the generic architecture of 'pluggable visual descriptions'. In this approach application managers control an application system by directly manipulating high-level domain concepts within a changeable set of visual descriptions. A set of novel assumption modules encapsulate those aspects of the domain that are

currently not enhanceable. These assumption modules, together with the visual descriptions, can be replaced to achieve meta-enhanceability. An expanded scenario is used to present ECHOES, which is an application of this approach to workflow systems.

Our experiences from implementing ECHOES are positive in regard to the functionality and expressive power of the proposed architecture and ECHOES. However, further research is needed to identify some usability characteristics of ECHOES, the applicability of the architecture for other application domains and the limits of meta-enhanceability for a particular domain model.

There are some important limitations to user enhanceable systems, which have been outlined in section 5. However, our limited experiences with ECHOES have supported the position [24, 21] that tapping into the domain expertise of the users can help to overcome most of these limitations, so that domain experts do not only achieve satisfactory results in software modifications, but they often tailor the software in novel and creative ways.

References

1. Jon A. Barrett. User Enhanceable Systems. Talk synopsis handed out on the User Enhanceable Systems : Enabling Technologies — JFIT preliminary proposals workshop, 18 Nov 1992, DTI, London, November 1992. Digital Equipment Corporation.
2. Jose A. Borges and Ralph E. Johnson. Multiparadigm visual programming language. In *Proceedings of the 1990 IEEE Workshop on Visual Languages*, pages 233–240, IEEE Service Center,Piscataway, NJ, USA, Oct 4-6 1990. IEEE Computer Soc and Univ of Pittsburgh and Knowledge Systems Inst and VL Foundation, IEEE. IEEE cat n 90TH0330-1, ISBN: 0-8186-2090-0.
3. Frederick P Brooks Jr. *The Mythical Man Month.* Addison-Wesley, London, 1975.
4. M. Burnett and A. Ambler. Interactive visual data abstraction in a declarative visual programming language. *J. Visual Languages and Computing*, 5(1):29–60, March 1994.
5. Margaret M. Burnett, Marla J. Baker, and Pieter van Zee. Scaling up visual programming languages. *IEEE Computer*, 28(3):45, March 1995.
6. S Ditlea. Spreadsheets can be hazardous to your health. *Personal Computing*, pages 60–69, January 1987. Cited in [26].
7. Jean Dollimore and Sylvia Wilbur. Experiences in building a configurable CSCW system. In J.M. Bowers and S.D. Benford, editors, *Studies in Computer Supported Cooperative Work*, pages 173–181. Elsevier Science Publishers B.V. (North-Holland), 1991.
8. Jr. F. P. Brooks. No silver bullet. *IEEE Computer*, 20(4):10–19, April 1987.
9. E.D. Falkenberg, J. L. H. Oei, and H. A. Proper. A conceptual framework for evolving information systems. In H. G. Sol and R. L. Crosslin, editors, *Dynamic Modelling of Information Systems, II*, pages 353–375. North-Holland, Elsevier Science Publishers B.V., 1992.
10. Paolino Di Felice, Frederick L. Lochovsky, and Thierry Mosser. Officeaid VPE: a visual programming with examples system for specifying routine office tasks. *Journal of Visual Languages and Computing*, 2(3):275–296, 1991.

11. Adele Goldberg and David Robson. *Smalltalk-80: The language.* ParcPlace Systems Inc., 1989.
12. Robert B. Grafton and Tadao Ichikawa. Visual programming (guest editors' introduction). *IEEE Computer*, 18(8):6–9, August 1985.
13. J.C. Grundy and J.G. Hosking. Constructing multi-view editing environments using MViews. In *Proceedings of the 1993 IEEE Symposium on Visual Languages*, IEEE Service Center,Piscataway, NJ, USA, August 1993. IEEE Press.
14. Masahito Hirakawa, Minoru Tanaka, and Tadao Ichikawa. An iconic programming system, HI-VISUAL. *IEEE Transactions on Software Engineering*, 16(10):1178–1184, October 1990.
15. Stefan Jablonski. Workflow management systems — modelling and architecture. Tutorial notes on CAiSE*94, June 1994. Utrecht, the Netherlands.
16. JFIT. User enhanceable systems : Enabling technologies. JFIT preliminary proposals workshop, 18 Nov 1992, DTI, London.
17. Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings OOPSLA '92, ACM SIGPLAN Notices*, pages 63–76, October 1992. Published as Proceedings OOPSLA '92, ACM SIGPLAN Notices, volume 27, number 10.
18. Phil Jones. Editorial. *Informatics*, 15(6):3, June 1994.
19. Gabor Karsai. A configurable visual programming environment: A tool for domain-specific programming. *IEEE Computer*, 28(3):36, March 1995.
20. M M Lehman and L A Belady, editors. *Program Evolution: Processes of Software Change.* Academic Press, London, 1985.
21. Wendy E. Mackay. Beyond iterative design: User innovation in co-adaptive systems. Technical Report EPC-1991-130, Rank Xerox Research Centre, Cambridge Laboratory, 61 Regent Street, Cambridge CB2 1AB, 1991.
22. V.A.J. Maller. User enhanceable systems. an outline proposal for a new DTI/SERC advanced technology programme in IT. distributed on User Enhanceable Systems : Enabling Technologies — JFIT preliminary proposals workshop, DTI, London, 18 Nov 1992.
23. Thomas W. Malone, Kum-Yew Lai, and Christopher Fry. Experiments with Oval: A radically tailorable tool for cooperative work. In Jon Turner and Robert Kraut, editors, *Proceedings of the ACM CSCW'92 Conference on Computer-Supported Cooperative Work: Emerging technologies for Cooperative Work*, pages 289–297, POB 64145, Baltimore, MD 21264, 31 October - 4 November 1992. ACM, ACM. ACM Order Number: 612920.
24. Bonnie A. Nardi. *A Small Matter of Programming: Perspectives on End User Computing.* Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1993.
25. D Novick and E Wynn. Participatory conversation in PD. *Communications of the ACM*, 36(4):93, June 1993. Note.
26. Raymond R. Panko. Introduction to the minitrack on risks in end user computing. In *Proceedings of the Twenty-Ninth Hawaii International Conference on System Sciences*, January 1996.
27. CCD Poo and PJ Layzell. Enhancing software maintenance through explicit system representation. *Information and software technology*, 32(3):176–186, 1990.
28. Georg Raeder. A survey of current graphical programming techniques. *IEEE Computer*, 18(8):11–25, August 1985.
29. Alexander Repenning and Tamara Sumner. Agentsheets: A medium for creating domain-oriented visual languages. *IEEE Computer*, 28(3):17–25, March 1995.

30. Nan C. Shu. *Visual Programming.* Van Nostrand Reinhold Company Inc., New York, 1988.
31. L. Slusky. Modelling of I-CASE platform. *Information and Software Technology,* 33(8):547–558, October 1991.
32. Keith D Swenson, Kent T Irwin, Robin J Maxwell, Toshikazu Matsumoto, and Bahram Saghari. A business process environment supporting collaborative planning. *Journal of Collaborative Computing,* 1(1):15–34, March 1994.
33. S. Tanimoto. Viva: A visual language for image processing. *Journal of Visual Languages and Computing,* 1(2):127–139, June 1990.