The Emerging VITAL Workbench[1,2]

John Domingue, Enrico Motta, and Stuart Watt

Human Cognition Research Laboratory
The Open University
Walton Hall,
Milton Keynes, MK7 6AA, UK
email: j.b.domingue@open.ac.uk

Abstract. VITAL is a research and development project which aims to provide methodological and software support for developing large, embedded KBS applications. VITAL is novel in that its ambition is to develop a methodology-based workbench covering the whole KBS life-cycle, from requirements specification to implementation, and to integrate and deploy a number of techniques drawn from artificial intelligence, as well as software engineering and human-computer interaction fields of research. In this paper we report on the current state of the VITAL workbench, and in particular we discuss the general design choices we took concerning the overall infrastructure, user interface, data and control integration, and tool management. Moreover, we'll describe in some detail the important role that some advanced software technologies - such as groupware and software visualization - have played in the design and implementation of the workbench.

1. Introduction

There is widespread consensus in the Knowledge-Based Systems (KBS) community that a more rigorous development process is required if large, embedded Artificial Intelligence (AI) applications can be successfully developed and deployed. Therefore, over the past few years there has been quite a lot of activity both in the States and in Europe aimed at producing more robust, structured, and reproducible methodologies and techniques for developing KBS applications. These initiatives include the KADS-II and VITAL ESPRIT-funded projects in Europe, and the Knowledge Sharing Initiative in the United States.

---

1 This paper appeared in Aussenac, N., Boy, G., Gaines, B., Linster, M., Ganascia, J.-G. & Kodratoff, Y. (eds) Knowledge Acquisition for Knowledge-Based Systems 7th European Workshop, EKAW'93 Toulouse and Caylus, France, September, 1993, pp. 320-339, Springer-Verlag.

2 Thanks go to Marc Eisenstadt and Blaine Price for providing descriptions of Viz, and to all our colleagues in VITAL for the numerous interesting discussions.

The VITAL project is a four and a half year research and development enterprise[3], involving nine organisations drawn from five countries[4]. VITAL aims to provide both methodological and software support for developing large, industrial, embedded KBS applications. The methodology, which specifies the VITAL approach to KBS development as well as to KBS project management, integrates methods and techniques from both AI and Software Engineering (SE), and aims to make the development of KBS applications more rigorous and structured. The main objective of the project, however, is the construction of the VITAL workbench, an integrated software environment supporting the VITAL methodology.

In this paper we describe the current state of the VITAL workbench, in particular illustrating the general design choices we took concerning the overall infrastructure, user interface, data and control integration, and tool management. Moreover, we'll describe in some detail the important role that some advanced software technologies - such as groupware and software visualization - have played in the design and implementation of the workbench.

Before embarking on a description of the workbench we will first provide a quick overview of the VITAL approach to KBS development, which, of course, provided the main input to the design of the workbench.

2. The VITAL Knowledge Engineering Methodology

2.1 Process Products

The VITAL knowledge engineering methodology is centred around the notion of a *process product*, which is an "essential and permanent deliverable produced in the course of a KBS project" [Jonker et al, 1991]. The idea here is that the development of a KBS is facilitated by the adoption of a structured approach, in which: i) the development of an application is guided by the construction of well-defined and well-documented process products, ii) the role of each process product and its links to the others are made explicit, and iii) there is a consistent and systematic set of techniques and methods that support the construction of each process product. A KBS project developed in accordance with the VITAL methodology should deliver the four process products listed below.

Requirements Specification (RS). This is a document providing a description of the expected functionalities of the application, and eventual constraints which need to be obeyed.

Conceptual Model (CM). This process product provides a *model of expertise* [Wielinga et al, 1992], capturing the relevant entities of the domain, task structures, and expert problem-solving behaviour.

---

3 VITAL is partially funded by the ESPRIT Program of the Commision of the European Communities, as project number 5365.

4 The partners in the VITAL project are the following: Syseca Temps Reel (F), Bull Cediag (F), Onera (F), The Open University (UK), University of Nottingham (UK), Royal PTT Netherland (NL), Nokia (SF), University of Helsinki (SF), and Andersen Consulting (E). This paper reflects the opinions of the authors and not necessarily those of the consortium.

Design Models (DM). These comprise the Functional Design Model (FDM), and the Technical Design Model (TDM). The FDM provides an implementation-independent description of the target KBS. The TDM is an implementation specific (it can also be domain and KBS specific) mapping between the FDM and Executable Code (EC).

Executable Code (EC). This comprises all 'maintainable software components' embedded in the application (whether or not they have been developed in the KBS project in question).

In the course of the past two years, a number of techniques, languages, and models have been devised to support the specification of process products. These include guidelines on how to structure a RS [Tang & Vuorenmaa, 1991], formal and informal languages for conceptual modelling [Koopman et al, 1991], the VITAL life-cycle model [Kontio & Rouge, 1991], the VITAL knowledge acquisition (KA) methodology [O'Hara et al, 1992], the VITAL approach to functional design [Jonker & Motta, 1993], a framework for implementing hybrid KBS [Motta et al, 1991a], the VITAL bug location methodology [Domingue et al, 1993], and techniques for verification and validation [Lapicque et al, 1991].

Of course, because of space limitations, is not possible to present a complete overview of the VITAL methodology here, and we'll instead emphasize some of the main ideas which have pervaded the shaping of the VITAL approach to KBS development.

2.2 Some Aspects of the VITAL Approach

Knowledge Engineering vs Project Management. The VITAL approach includes a knowledge engineering and a *project management methodology.* The former describes, for each process product, its role in the KBS development process, its structure and semantics, the modelling language used to represent its contents, the process by which the process product in question is developed, and what techniques and tools can be applied to support its development. The latter defines a number of *life cycle configurations*, which specify how the development of each process product takes place within the overall management of the application project. In particular, a life cycle configuration provides a mapping between *management phases*, which are part of the project management methodology, and the components of the knowledge engineering methodology. In general, a life-cycle configuration for a particular project is chosen taking into account the type, size, cost, and risks inherent to the project in question.

Process Products as a Mechanism for Organizing Development and Maintenance of KBS. We have already mentioned that the main idea of process products is to make sure that the complex task of developing KBS applications is decomposed into a number of more manageable and qualitatively distinct activities. Another advantage of the process product paradigm is that it can be used to facilitate the maintenance of large applications. During the development of a KBS, semi-automatic transformations take place between the process products. The VITAL tools supporting this process keep 'historical dependency' links between the connected parts. These links enable the user to see where a particular process product entity came from (and why it was created), and can be thought of as the 'design rationale' for the KBS implementation. This information can then be used in the maintenance phase of the life-cycle. For example, when the root cause of a bug has been tracked down within the EC process product, the appropriate part of the upstream models can be brought up and inspected. This inter-process product bug tracking is a main component of the VITAL bug location methodology [Domingue et al, 1993].

Knowledge-level Models. One important aspect of a KBS methodology is the provision of techniques, methodologies, and languages for analysing domain data and expert problem solving behaviour in an implementation-independent way. This *knowledge-level modelling* perspective, which originates from the influential paper by Newell [Newell, 1982], is now well established and has been used in a number of approaches to knowledge acquisition and KBS design [Steels, 1990; Wielinga et al, 1992; Clancey, 1992; Chandrasekaran et al, 1992]. The main advantage of this kind of approach is that it enables practitioners to acquire, model, analyse, validate, and compare models of expertise only in terms of the modelled knowledge, without the 'noise' produced by the implementation-level structures used to encode the knowledge, say, in a programming language, or an expert system shell.

Both the conceptual and functional design models in VITAL are knowledge-level descriptions. The former provides a model of the domain in hand, describing the relevant domain entities - i.e. the important concepts and relations in the domain - and the procedures used by domain experts - in particular the task decomposition which experts use during their problem solving. A functional design model is instead an abstract specification of the performance system. Sometimes it can be the case that there is little or no difference at all between a conceptual and a functional design model. For instance, the specification of the Sisyphus I problem [Linster, 1992] requires the target system to mimic the behaviour of the domain expert in question, and in [Motta et al, 1993] we describe a VITAL-based solution to the problem in which the functional design model is nothing more than a fine-tuning of the associated conceptual model. However, this was made possible by the fact that i) there was the explicit requirement that the solution had to be a cognitive model of the expert, and ii) there were no 'deployment-related' requirements, such as constraints on the IO modality or on the expected performance, which would have required a more radical differentiation between the model of the problem and the system specification. Of course, even if one accepts that these two models play different roles in the VITAL framework, one could ask "why bother with conceptual modelling at all?". The answer to this question is that this separation has a number of advantages:

- It enables the knowledge engineer to model the domain in hand without worrying about a number of issues which are only relevant at the design stage. These include: fine-grained control decisions, efficiency aspects, and the specification of those tasks, typically explanation and IO tasks, which are only defined to meet system requirements.
- It enables the knowledge engineer to develop problem-solving models which differ from those employed by experts, while keeping track of both models.
- Provides a natural task decomposition for a KBS project, where knowledge acquisition and conceptual modelling are carried out as 'autonomous' enterprises, distinct from system specification. This is consistent with i) the fact that many KBS projects make use of specialized knowledge analysts, who are experts in knowledge acquisition, but do not normally design and implement systems; and ii) the fact that in some cases conceptual modelling is the main goal of an AI project. This is the case when a project is set up

only to record and formalize specific skills in an organization, or when domain-specific, but application-independent ontologies are devised.

Because both the functional design and the conceptual model are knowledge-level specifications, and because in general a lot of the conceptual model is reused during functional design, the same formalism and methodology is used for both kinds of models. Thus, in the rest of the paper, we will use the term 'knowledge-level model' to refer to a generic conceptual or design model.

Formal and Informal Languages. In VITAL we provide both formal and informal languages for representing knowledge-level models. As pointed out in [Jonker et al, 1991], "formal approaches have a great potential to improve the quality of the KBS development...In particular, they can provide support for correctness-preserving transformation". On the other hand, there can be situations in which a formal approach to knowledge-level modelling might not be appropriate. This can be the case if the knowledge modellers in question do not possess the required formal skills, or if the cost of producing a formal specification offsets the resulting benefits.

An alternative approach to producing non-ambiguous specifications, which can be applied in situations in which formal approaches are not suitable, is through the development of operational, knowledge-level models. These are models specified in a language for which an interpreter has been defined. In addition to providing the implementor with a non-ambiguous specification, operational models are also useful as they enable knowledge modellers to test and evaluate the model.

Thus, we have in VITAL two languages for representing conceptual and design models. The first one, the VITAL Knowledge Modelling Language, VITAL-KML[5], has been designed with the idea of supporting quick development of conceptual and design models, and rapid prototyping of applications, as it can be operationalised by means of a compilation process to the VITAL-KR [Motta et al, 1991a][6].

The other modelling language, K_{bs}SF, intends to support those designers who need a more formal[7] specification of a model. This second language integrates Algebraic Data Types, order sorted logic, and a formal representation for tasks.

Of course, the two languages have mutually consistent structures and it is possible to map the VITAL-KML into K_{bs}SF specifications. The specifications of these two formalisms can be found in [Jonker & Motta, 1993].

---

5 The VITAL-KML is an extension of an earlier, informal language - VITAL-CML -, which was developed to support the construction of informal conceptual models [Koopman et al, 1991]. In order to emphasise i) the derivation from the VITAL-CML, and ii) the fact that it is an operational language, the VITAL-KML is sometimes referred to in VITAL documents as the 'VITAL-OCML' (operational CML). However, we prefer to call it 'VITAL-KML' to emphasize that it can be used for knowledge-level modelling at both the conceptual modelling and design level.

6 This is a tool for integrating heterogeneous reasoning components, which provides the core of the implementation-level support in the VITAL workbench.

7 The use of the terms 'formal' and 'informal' here is a little bit artificial. In fact it can be argued that because the VITAL-KML can be operationalized, it is not really 'informal'.

2.3 Advantages of the VITAL Approach

To summarize the discussion, the VITAL methodology consists of a structured development process, during which the four process products are created. This approach provides a number of advantages, such as:

i) it provides the structure for effective life-cycle management;

ii) it explicitly distinguishes between the specification and the implementation of a system, therefore facilitating verification and testing of the application;

iii) it facilitates system maintenance, as changes can be introduced at anyone of the four different levels;

iv) it facilitates debugging, as the cause of a problem at implementation level can be traced back to errors in the specification of the system, or in the domain model;

v) it facilitates vertical reusability, as anyone of the four process products (or any part thereof) can be reused for a different application.

With this we conclude the discussion concerning the VITAL approach to KBS development, and we can now turn our attention to the VITAL workbench.

3. Main Issues in the Design of the VITAL Workbench

The VITAL workbench is an integrated software environment which aims to support the various activities, models, and languages which are defined in the VITAL methodology. Whilst planning the construction of the workbench we have focused on three general[8], key questions.

- What design solutions would best support the KBS development process, given the process product centred framework used in VITAL? Is there a natural organisation of the workbench with respect to the VITAL approach? What are the implications of supporting the whole knowledge engineering process?
- What sort of software infrastructure is required for the creation of an integrated workbench? Given that the workbench will be a large software product which aims to integrate a number of different tools, the integration problems are not trivial.
- What new software technologies - for instance those developed within the Human Computer Interaction (HCI) community - could be incorporated into the VITAL workbench?

In the rest of this section we illustrate our answers to the above questions.

3.1 Assistants in the VITAL Workbench

The workbench mirrors the decomposition of the VITAL methodology, as it is functionally organised into four *assistants*. Each assistant comprises a set of tools which can be used for producing the relevant process product. This decomposition is of course functional rather than structural, as typically there is quite a lot of overlap between tools in different assistants. For instance, conceptual modelling is supported

8 Of course, there were a plethora of more specific questions (i.e. at the process product level) which we also had to face. For instance, deciding which knowledge acquisition tools should be integrated in the workbench [O'Hara et al, 1992].

by means of a knowledge editor, knowledge acquisition and machine learning tools, and verification and validation (V&V) tools. However, because the same formalism is used for both conceptual and design models, the same editor and the same V&V tools are actually used in both assistants.

The decomposition of the workbench into four assistants also had implications for deciding what the interface to the workbench should look like. The metaphor should naturally map into the assistant/tools organisation, provide an effective way of 'driving' the workbench, and allow easy customisations of the workbench in response to alternative life-cycle configurations. The answer to this problem is given by the *VITAL Tower*. This is a tool management tool which enables the user to browse, select and load tools in VITAL. The metaphor used in the VITAL Tower is that of different *rooms* and *floors*. In the simplest case a floor corresponds to an assistant, and the associated rooms to tools or set of tools which are included in the assistant in question. The floors/rooms metaphor provides a mechanism enabling users to customise their software environments, and, at the same time, a sort of 'control blackboard' for loading and selecting assistants/tools. Although the default set-up is one in which floors map to assistants and rooms to sets of tools which are part of the same assistant, the user is free to specify alternative configurations, which do not necessarily have to correspond strictly to an assistant-tools decomposition. The VITAL Tower is described in detail in section 4.

3.2 The Infrastructure of the VITAL Workbench

In addition to the issue of the workbench management tool, we needed to make, both at assistant and tool level, a number of decisions concerning control, data, and user interface integration of different assistants/tools in the workbench. The solutions we took are briefly described in this section. Figure 1 below summarises the organisation of the workbench.

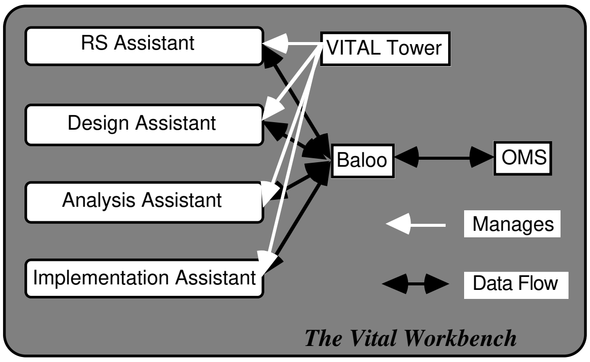

Figure 1. An overview of the organization of the VITAL workbench.

Control Level Integration. In general, the user will be free to use each assistant independently of the others, so that different kinds of users and process models can be supported (i.e. rapid prototyping as well as structured development). Therefore no control-level integration is being imposed at the assistant level. However, within a particular assistant, there may be specific control constraints.

Data Level Integration. Data integration is a difficult issue as it is strictly related to the choice of an Object Management System (OMS). We have tackled this by developing a *meta-object-management-system*, called *Baloo*, [Watt et al, 1992] which can be viewed as a mediating interface between a tool in the workbench and an OMS. Baloo enables tool developers to handle data storage at a level which is independent from a particular OMS. The advantage of this solution is that it enabled us to carry on with the design and development of the workbench without having to worry about or being constrained by issues related to the choice of an OMS. In particular, the first prototypes of the workbench have made use of simple, file-based data sharing and storage facilities, while at a later stage of the project Baloo will be interfaced with more powerful OMSs such as PCTE [Brown, 1991]. Because tools do not interact directly with the OMS, and because the interface between Baloo and a 'generic OMS' has been clearly specified, moving from one OMS to another will not require modifications to the VITAL tools.

User Interface Integration. This is addressed by the provision of:

- a common look and feel of the tools in the workbench;
- user interface (UI) design guidelines;
- a consistent interface to the different model building activities;
- a consistent interface to displaying dynamic systems;
- a general navigation and presentation metaphor for the workbench as a whole.

These are discussed in the next sections.

Common Look and Feel of the Tools in the Workbench. The default look and feel of the workbench is based on OSF Motif™ as this is the *de facto* standard for Unix machines.

User Interface Design Guidelines. The UI design guidelines [Watt et al, 1992], which are mainly an adaptation and integration of some well-known proposals [Apple, 1987; Schneiderman, 1987; Open Software Foundation, 1990], provide an interface 'philosophy', for the VITAL tool developers. These guidelines are quite extensive ranging from a classification of the various types of 'alert' messages, to the style of menu items, to advice on how to design modeless interfaces.

A Consistent Interface to the Different Model Building Activities. This aspect concerns the problem of ensuring the UI coherence of the environments supporting the various model building activities. Although the provision of a house style and a number of design guidelines should discourage tool developers from following wildly diverging approaches, these conditions alone do not guarantee the ultimate coherence of the various model editors in VITAL. Therefore, VITAL provides support in the form of a generic model editor, *Dale*, which can be tailored for each of the four process products. Dale consists of a set of interfaces which can be easily customised and extended. These are listed below:

- Grid Manager (or Spreadsheet Manager) for displaying tables.
- Graph Manager for displaying graphically entities and relations in a model.
- Text Manager for displaying text versions of model entities.
- Property Sheet Manager, to allow the properties of model entities to be both viewed and changed.

The idea here is that the various knowledge editors will be built as customisations of Dale. An example of a typical VITAL tool developed on top of Dale is provided in sections 6.1.

A Consistent Interface to Displaying Dynamic Systems. A parallel problem to that addressed by Dale is that of providing *visualizations* for the dynamic systems within the workbench. By 'visualization' here we mean the use of visual (and other) media to enhance one's understanding of a dynamic model.

The VITAL workbench will include quite a few dynamic systems, including knowledge acquisition (KA) and machine learning tools, V&V tools, rule interpreters, object-oriented systems, etc. These tools will manipulate and communicate large amounts of data. It is very important therefore, in order to make the workbench usable, that good visualization techniques are incorporated in VITAL, to enable the users to 'be in control' of the dynamic behaviour of the workbench tools. Because of the importance we associate to this problem, quite a number of resources have been allocated to developing a general framework and software library which provides a set of high level constructs to be used by the VITAL tool developers to construct their visualization systems. This framework is called *Viz* [Domingue et al, 1992], and is described in section 5.

A General Navigation and Presentation Metaphor for the Workbench. As already mentioned, this is provided by the VITAL Tower, which is described in the next section.

As discussed here, the three main tools which shape the infrastructure of the workbench are the VITAL Tower, the generic editor Dale, and the visualization support, Viz. These are described in the next three sections.

4. The VITAL Tower

4.1 Background

The mid-1980s witnessed a number of attempts to help users manage the desktop interface. The main problem was how to configure and manage a large number of windows dedicated to several tasks. One approach employs a 'virtual desktop' whereby the desktop can be infinitely extended in all directions [Bolt, 1984]. The user moves around and navigates this 'infinite' desktop using a coarse-grained 'bird's eye' view. Another approach [Henderson & Card, 1986] splits the interface into multiple and separate 'desktops' or 'rooms', each designed to handle a different kind of work. In general, a room only contains the data needed for that part of the work, although it is possible to share data between rooms. Navigation between rooms is enabled by an overview, called a 'stampsheet', which contains iconic representations of all of the rooms.

More recent research has focused on the management of KBS interfaces. Most currently available KBS development environments (e.g. Nexpert Object™, KEE™, ProKappa™) use a standard WIMP interface; however, these provide limited help with managing the tools within the system. There are a few exceptions to this. For example, KEATS [Eisenstadt et al, 1990; Motta et al, 1991b] makes use of a desktop interface called a *Dynaboard.* This consists of a panel of saved desktops, each represented by an icon which is a scaled screen image. Selecting a panel within the dynaboard switches to the saved desktop. Another system, Cyc [Lenat & Guha, 1990], uses a metaphor which maps frames into floor plans of rooms, each room representing one frame. The floor plan representation of connected rooms can then be browsed using a tool called the *Museum Interface.*

The contribution made by the VITAL interface is to provide a tool configuration and navigation facility which can easily be linked to a knowledge engineering methodology. This is not present in any of the above: no specific software engineering methodology is prescribed with the ‘Rooms’ metaphor; and, the navigation devices provided by KEATS and Cyc are not linked to their KBS development methodologies.

4.2 The Interface Requirements

The design of the VITAL Tower was driven by four major requirements:

- It should naturally link with the VITAL knowledge engineering methodology.
- It should allow the user to navigate easily through the large number of tools contained within the workbench.
- It should be compatible with the groupware facilities - see section 6.2.
- It should be adaptable so that any organisation could customise the workbench to meet its particular needs. This might be the case where, for example, an organisation has its own internal software management life cycle.

4.3 The VITAL Tower Metaphor

The metaphor has the following main components:

i) a *tower* - the interface to the workbench;

ii) a *floor* - the interface to a single assistant or set of tools external to VITAL;

iii) a *room* - a set of tools within an assistant that are used together to create a specific part of a process product.

Each room is devoted to a particular task. Rooms on the same floor are related by the fact that they operate on the same process product.

The metaphor promotes the following method of work: the user moves between rooms on a floor when he or she wants to view a portion of a process product in a particular way, or with a particular focus; the user moves to a new floor when he or she wants to *transform* the data of one process product into the form of another process product. The metaphor can be displayed on screen in either of two forms: a *Lift Panel* or a *Tower Plan*.

4.4 The VITAL Tower Lift Panel and the VITAL Tower Plan

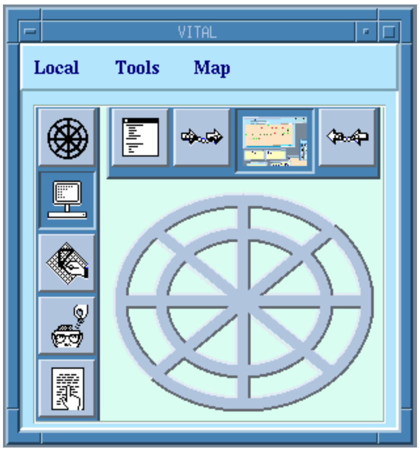

Figure 2. The VITAL Tower Lift Panel.

The VITAL Tower lift panel was designed to simulate a lift panel (containing the numbered floor buttons) in a real lift (elevator). The lift panel shown in Figure 2 consists of two sub-panels:

(i) a vertical 'lift button panel' displaying each floor as an icon; and

(ii) a horizontal 'current floor - rooms panel' showing all the rooms on the current floor.

The first four floors (in ascending order) map onto the four assistants. As in a real tower, the first floor is situated at the bottom of the lift button panel. The top floor contains rooms for carrying out external tasks such as mailing.

Selecting a floor button updates the current floor rooms panel, thus allowing the user to browse the rooms on that floor. A room can be visited by simply pressing the appropriate room panel button. When a room is visited for the first time the appropriate tools are automatically loaded and set up in a pre-determined state. The iconic representation of the room is then replaced with a scaled screen snapshot which is updated on leaving the room. As a result it is possible to tell, from the lift panel, which rooms on the current floor have and have not yet been visited, i.e. which tasks have yet to be tackled.

The VITAL Tower Plan allows all the rooms to be inspected simultaneously. Figure 3 shows a section of the plan after two rooms (the larger icons) have been visited.

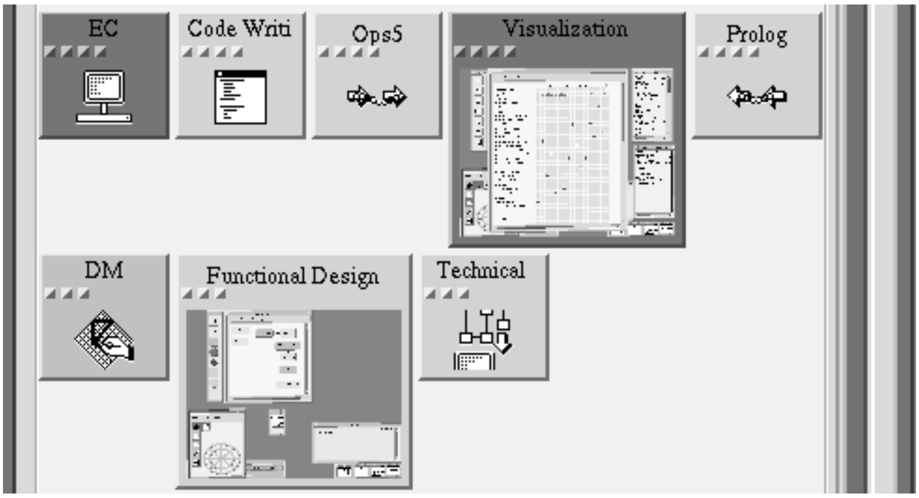

Figure 3. A section of the VITAL Tower plan.

The floor icons are presented down the left side of the plan and the rooms on each floor are displayed to the right of the floor icon. The nodules on each floor or room icon indicate the floor number. The rooms on a particular floor can be displayed or hidden by double-clicking on the floor icon. Just as with the lift panel, the plan can be used to visit rooms by simply clicking on the room. Again, once a room has been visited the iconic representation is replaced with a scaled snapshot of the current screen.

A new room may be created by duplicating the current room and then setting the new room's state by opening the desired tools. The new room is placed next to the current room within the plan and lift panel.

The VITAL Tower fulfils the interface requirements in the following ways:

- It supports the VITAL methodology - the link between the assistants and the floors maps directly onto the methodology.
- It allows the user to navigate a large number of tools – the grouping of tools into task related rooms and the grouping of rooms into assistant related floors provides an abstraction mechanism allowing a very large number of tools to be managed.
- It is compatible with the groupware extension. As pointed out in [Stefik et al, 1987], "We hypothesise that WYSIWIS Rooms will prove increasingly effective with larger groups and larger tasks – just as larger physical rooms and breakout rooms are advocated in meetings not supported by computers".

The key contribution made by the VITAL Tower is that it allows users to configure their work at two different levels. At a methodological level the major phases of a particular methodology can be used to determine what floors should be present within a tower, and the tasks within a phase would determine the rooms within each floor. At the tool level, each room can be set up according to the task to be tackled and the individual user's preferences.

5. Software Visualization in VITAL.

There are two ways in which visualization techniques can facilitate the process of understanding dynamic models, such as those which are developed in a typical KBS application. Firstly, they can be utilised to create a single coherent execution model (story) from the heterogeneous inference steps, thus enabling an overview of the dynamic behaviour to be obtained.

Secondly, visualizations can display information in a non-linear fashion. Items can be located directly, rather than by a linear search, and patterns can be found by gestalt. Because we have deemed that visualizations are an important technique to be used whenever necessary, we envisage that a number of them will be constructed for the workbench. In order that this set could be built without undue overhead in the project, we have created a framework and software library, called Viz, of which we shall now give a brief overview.

5.1 Overview of Viz.

Viz's main contribution is that it provides the VITAL developer with a very high (programming) language for constructing as well as describing visualizations. Thus, from a design point of view, it is possible to develop more sophisticated and principled visualizations, while, from an implementation point of view, it is much quicker to transform these high level designs into running systems.

The overarching metaphor used in Viz is that of 'story-telling', which considers program execution to be a series of history events happening to (or caused by) players. To allow our 'story-tellers' (VITAL system builders) considerable freedom, a player can be any part of a program, such as a function, a data or knowledge representation structure, or a line of code. Each player has a name and is in some state, which may change when an event occurs for that player. These events are recorded in a history module, which allows them to be accessed by the system builder and 'replayed'. Events and states are mapped into a visual representation which is accessible to the end-user (the programmers who need to use the visualization system, not the visualization system builder). But the mapping is not just a question of storing pixel patterns to correspond to different events and states—we also need to specify different views, and ways of navigating around them. This framework leads to following set of primitive Viz constructs:

- *Histories*: a record of key events that occur over time as the program runs, with each event belonging to a player; each event is linked to some part of the code and may cause a player to change its state.
- *Views*: the style in which a particular set of players, states or events is presented, such as using text, a tree, or a plotted graph; each view uses its own style and emphasises a particular dimension of the data that it is displaying.
- *Mappings*: the encodings used by a player to show its state changes in diagrammatic or textual form on a view using some kind of graphical language, typography, or sound; some of a player's mappings may be for the exclusive use of its navigators.
- *Navigators*: the tools or techniques making up the interface that allows the user to traverse a view, move between multiple views, change scale, compress or expand objects, and move forward or backward in time through the histories.

5.2 A Sample Visualization

As an example, we show in figure 4 a visualization, defined in table 1, of the execution space of a sample rule interpreter run.

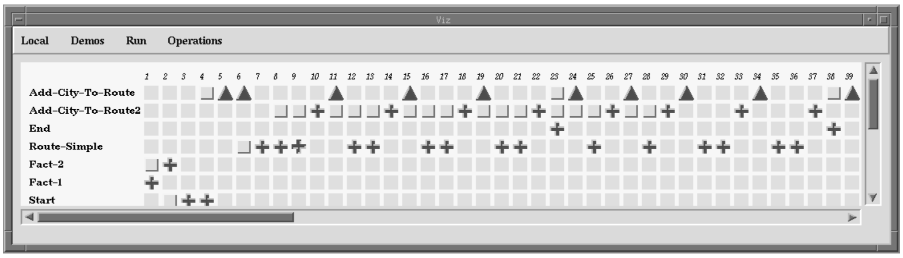

Figure 4. A screen snapshot of the rule interpreter visualizer.

	Viz Definition
Players	rule, rule instantiation; *the rule player is composed of rule instantation players*
States	failed to match working memory; *applies only to rule players* matched against working memory; *applies only to rule players* fired; *applies to rule and rule instantiation players* fired and invoked the backward chainer; *applies to rule and rule instantiation players*
Events	matching against working memory; firing; invoking the backward chainer
Mappings	failed to match working memory -> matched against working memory -> fired -> invoking the backward chainer ->
Views (in decreasing order of granularity)	table: current state; *in a table based view display the rules current state* text: rule's instantiation states; *in a text based view show the rules instantiations states*

Table 1. A summary of the TRI viz definition.

In the visualization shown in figure 4, based on the Transparent Rule Interpreter (TRI) [Domingue & Eisenstadt, 1989], a triangle indicates that a rule fired and invoked the backward chainer; a cross indicates that a rule fired; a shadowed box indicates that a rule matched against working memory, creating one or more instantiations, but did not fire; and a box indicates that the rule failed to match working memory.

In the visualization definition, shown in table 1 (comments are in italics), the rules and rule instantiations are defined as players. During each TRI match/fire cycle, the 'matching against working memory' 'firing' and 'invoking the backward chainer' events cause the rule and rule instantiations to change to the appropriate state. The players, events, and states for a particular run are stored by Viz's history mechanism. When a visualization of a run is requested, the stored history is displayed using the defined view, a table, and the above mappings. If the user clicks on one of the cells in the table (i.e. one of the mappings), a fine-grained view is displayed, in this case the text based one. This view provides a finer-grained level of visualization as it allows all of a rule's current instantiations to be displayed.

The Viz framework and software have been tested by creating a number of visualization systems which are now part of the workbench. In addition it will also provide the core of the VITAL debugging support [Domingue et al, 1993].

6. Dale, a Generic Model Editor

As discussed in section 3.2, VITAL provides a generic editor - Dale - which can be customized to provide graphical and text editing capabilities for the various modelling languages used in VITAL. In this section we illustrate one such customization of Dale, which provides an integrated graphical and text editor enabling users to create, modify, and browse knowledge-level models specified in the VITAL-KML. The details of this formalism can be found in [Jonker & Motta, 1993]. Here it suffices to say that it supports the specification of concepts, rules, relations, tasks, control and data flow diagrams.

6.1 Editing Knowledge-level Models using Dale.

There are two main ideas which have been incorporated in Dale, and therefore in all Dale-based tools. The first one concerns the advantages of the graphical representation of knowledge, which facilitates knowledge acquisition and validation [Gaines, 1991]. The second one is the need for integrated text and graphics in order to support effective model-building activities. In certain cases - for instance knowledge acquisition, diagrammatic specification of control, and browsing - graphical representations are more effective that text-based editors. In other cases, however, for instance when entering or modifying a large amount of data, it is much quicker to use a 'traditional' text editor. Therefore, we have attached great importance to the integration of these two forms of interface, in order to provide effective support for model building activities.

Figure 6 shows a data flow diagram which is part of the design model produced for the Sisyphus example. On the left hand menu panel there are a series of icons which map onto various drawing tools at the users disposal. From the top, they allow users to draw data flow (dashed arrow) or control flow (ordinary arrow) relations, specify a task, add one of the control constructs - SEQ (double oval icon) or CHOICE (diamond icon) - or add a concept, a concept instance, or a model[9].

[9] As in the COMMET workbench [Steels, 1992] the term 'model' is used here to specify the input or output to a task.

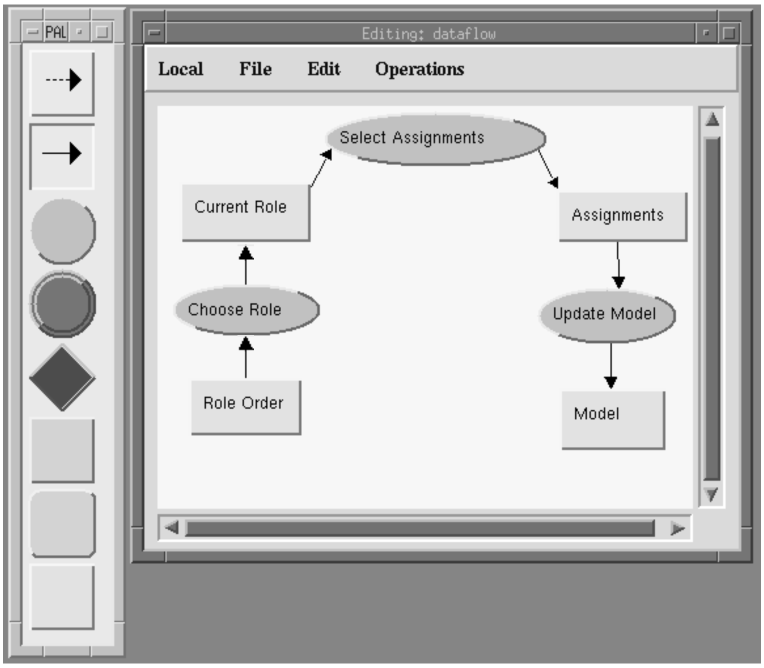

Figure 5. A screen snapshot of the Knowledge Modelling Tool displaying a data flow diagram. Ovals represent tasks, while boxes represent models.

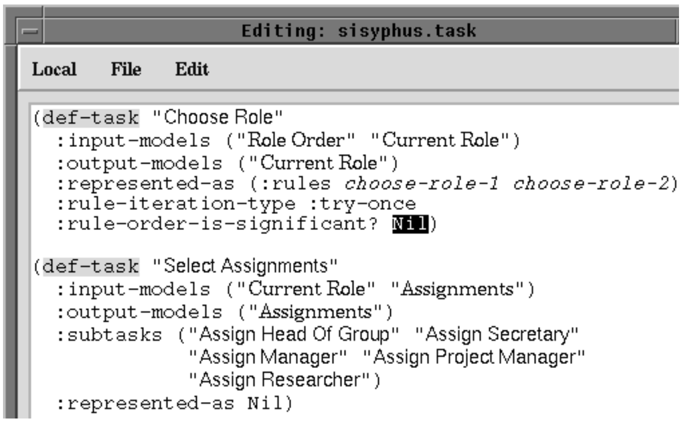

Figure 6. A screen snapshot of the text part of the knowledge editor.

In order to support this close text-graphics integration, every graphical component is associated with a textual counterpart. Changes to the text are reflected in the graphical view and vice-versa. This means that knowledge engineers can mix textual and graphical interaction styles to suit their needs for the task at hand. The text editor contains an incremental parser which both formats the code, in a style similar to that used by [Baecker & Marcus, 1990] for C programs, and acts as a syntax prompt. Figure 6 above shows a screen snapshot of the text editor displaying two of the tasks contained in figure 5.

Creating graphical and textual editors such as the knowledge modelling tool are relatively easy in Dale. For each entity (such as a task or model) the system builder specifies the following information:

- what the graphical representation (e.g. a red circle) should be (a library of customisable graphical objects is provided by Dale),
- what entity types (e.g. models and tasks) it can be connected to,
- what sort of link (plain, dashed, or with an arrowhead) should be used in the connections,
- what operations should be available from the operations menu,
- what the textual format is - including the structure of a new form (e.g. which parts are fixed parameters and which will need to be edited),
- how a textual form is to be parsed,
- what text style (family, face, size, and background and foreground colour) should be used for each part of the textual form,

Once the above has been specified Dale takes care of the book-keeping activities (e.g. making sure the graphics and text are always in step) and low-level graphical operations (e.g. scrolling, refreshing, opening and closing sub-windows).

We can see from the figure 6 above that in the knowledge text editor the following textual format has been defined:

- the default font is Courier,
- defining form names, such as def-task, have a light grey background colour,
- task names are shown in Helvetica font, model names in New Century Schoolbook font, and rule names in an italic style.

As shown here, the VITAL tool builder can develop customized Dale editors for specific modelling languages with relatively little effort. Another advantage of a Dale-based editor is that it can support collaborative model editing, as described in the next section.

6.2 Groupware Support through Dale.

An essential aspect of a methodologically-founded workbench is its integration with its users and their organisation. If a computer system is applied without care it can seriously damage an organisation by increasing centralisation and making essential skills inaccessible [Clement & Gotlieb, 1987]. Computer-supported cooperative work (CSCW) can help overcome these problems by studying how people, organisations, and systems interact. One kind of CSCW, 'groupware' has been defined as "computer-based systems that support groups of people engaged in a common task (or goal) and that provide an interface to a shared environment" [Ellis et al, 1991]. It is clear that a workbench such as VITAL – which is intended to assist the development of quite large systems – can only be fully utilised if it can support group or team work effectively.

Groupware systems can be broadly divided into two kinds, real-time and non-real-time. In the first, changes happen 'immediately' for all participants; in the second, changes are saved and get merged with some kind of conflict resolution, usually overnight. Real-time systems, the graphical implementations of which are called WYSIWIS (What You See Is What I See), seem to be important to keep a group focus. Note that this approach helps users see each others intermediate results, which seems to be quite important. Users do not only want to see consistent results, but also inconsistent ones as part of the development process.

Real-time groupware systems let many people edit the same data at the same time. Each participant can see the changes that the others are making. The tools in the VITAL workbench will provide this kind of group operation, so that several engineers can work on the same data at the same time. This is achieved through the generic editor, Dale, and the common data storage system, Baloo, where Dale acts as a generic graphical editor for objects represented in Baloo. The groupware system operates by 'connecting together' the Baloo objects, so that changes made by one participant are automatically forwarded to everyone else using that object at the same time, and so that everyone's view of the object is consistent at all times. Once the Baloo objects are maintained as shared by the group, the design of Dale ensures that each participant's graphical views are consistent, so implementing WYSIWIS groupware.

Conflict resolution is implemented by a variant of the dOPT algorithm [Ellis & Gibbs, 1989] which maintains consistency by modifying a participant's commands in a conflict so that the effects of the commands are the same, even though, because of delay in groupware communication, they may be applied in a different order. This avoids needing to lock objects to a single writer at a time, which can be slow, and also frustrating for other participants. Instead, the semantics of the commands are carefully defined so that their potential conflicts can be recognised and the commands modified accordingly. The design of Baloo assists in this, by defining a minimal set of commands and by defining their semantics precisely.

The principles and concepts of groupware provide a useful set of design rules which are shaping the development of the VITAL workbench. Real-time groupware will help create a consistent group focus for the users, and its implementation through Dale and Baloo ensures that it can be applied easily to new editors and object management systems without needing to extend the system. The VITAL Tower illusion works well with groupware, as the rooms provide a natural grounding for real-time collaboration. Rooms have the advantage that they can be designed by the people who are going to use them, rather than being rigidly enforced by the system. This flexibility helps the VITAL workbench to overcome many of the potential problems inherent in a computer system intended to support a methodology involving many people in many different roles. Groupware within VITAL is described in more detail in [Watt, 1993].

7. Related Work

The VITAL workbench is pretty unique in terms of coverage of the spectrum of activities related to KBS development. While other large workbenches have been developed, which integrate a number of tools and support aspects of a coherent methodology, for instance Shelley [Anjewierden et al, 1992a], and KEW [Anjewierden et al, 1992b], these only provide support for the upstream phases of knowledge engineering. In addition, while they make use of some 'advanced' software technologies, for instance hypertext, they put less emphasis on the aspects related to tool management, software visualization, and groupware. We should also add that a number of knowledge acquisition and machine learning tools from KEW are being integrated into the VITAL workbench.

The COMMET workbench [Steels, 1992] supports both the design and implementation of KBS. Whilst the design and implementation levels are integrated, the integration is based on files and is therefore not as tight as in VITAL. Moreover it does not support groupwork, nor visualization.

A similar approach to the one we have taken in VITAL has been adopted in the sets of tools supporting the Cyc project [Lenat & Guha, 1990]. Again, while the scope of their environment is not as broad (in terms of life-cycle coverage) as ours, they also make full use of visualization and groupware technologies, and they give particular importance to the problem of navigating large sets of data. In contrast with Cyc, we think that graphical visualization techniques can scale-up to handle large sets of data, provided that the right abstraction mechanisms have been defined. Cyc also provides groupware support which is non-real-time. In VITAL we also plan to provide support for real-time groupware.

A number of existing architectures have influenced the design of the VITAL workbench. Our own KEATS toolkit [Eisenstadt et al, 1990; Motta et al, 1991b] can be regarded as the predecessor of the VITAL workbench, and has therefore influenced our work most. While KEATS doesn't offer the same comprehensive methodology as VITAL, and only integrates a small number of tools, it has provided extensive input to VITAL. In particular, the VITAL knowledge representation architecture, the debugging, visualization, and tool management systems are direct descendants of the KEATS knowledge representation formalisms, visualization support [Domingue & Eisenstadt, 1989; Eisenstadt et al, 1990], and the KEATS tool management tool - Dynaboard [Eisenstadt et al, 1990].

8. Conclusions

In this paper we have provided an overview of the VITAL workbench, in particular discussing its overall organisation, the main design issues faced, and the solutions which have been given to them. We also described three completed tools.

The particular strengths of the emerging VITAL workbench are as follows:

i) It closely tracks related developments (e.g. KADS-II, KEW, COMMET, Cyc), incorporating relevant results and methodologies where possible.

ii) It goes beyond other approaches by covering the whole KBS life-cycle, from requirements specification to maintenance.

iii) It provides a novel integration of the upstream process products with a variety of knowledge representation formalisms at the implementation level [Motta et al 1991a].

iv) It provides advanced program visualization and visual programming technologies across a wide range of workbench tools.

v) It provides groupware support that is smoothly integrated into the core of the tools through the infrastructure of Dale and Baloo.

The VITAL project tries not just to tackle a specific phase of the knowledge engineering life-cycle, but it attempts to provide a comprehensive methodological and software framework, covering the whole spectrum of software development. We are now two and a half years into the project, and almost all the pieces of the puzzle are now in place. The next two years will be crucial, as the methodology and workbench will be tested on a number of real-world problems.

References

Anjewierden, A., Wielemaker, J., and Toussaint, C. (1992a) Shelley - computer aided knowledge engineering. Knowledge Acquisition. Special issue: “The KADS approach to knowledge engineering” 4(1). pp. 109-125.

Anjewierden, A., Wielinga, B., and Shadbolt, N. (1992b) Supporting Knowledge Acquisition: The ACKnowledge Project. In Esprit-92 Knowledge Engineering (eds. L. Steels and B. Lepape).

Apple Computer, Inc. (1987) Human Interface Guidelines: the Apple Desktop Interface. Addison-Wesley.

Baecker, R. M., and Marcus, A. (1990) Human Factors and Typography for More Readable Programs. Addison-Wesley.

Bolt, R. A. (1984) The Human Interface. Lifetime Learning Publications, Belmont, California.

Brown, A. W. (1991) Integrated Project Support Environments: the ASPECT Project.

Chandrasekaran, B., Johnson, T. R., and Smith, J. W. (1992). Task-Structure Analysis for Knowledge Modelling. Communications of the ACM 35(9).

Clancey, A. (1992) Model Construction Operators. Journal of Artificial Intelligence 53(1).

Clement, A., and Gotlieb, G. C. (1987) Evolution of an Organizational Interface: The New Business Department of a Large Insurance Firm. ACM Transactions on Office Information Systems 5(4), pp. 328-339.

Domingue, J. B. and Eisenstadt, M. (1989) A New Metaphor for the Graphical Explanation of Forward Chaining Rule Execution. Proceedings of the 11th International Joint Conference on Artificial Intelligence (IJCAI’89) San Mateo, CA. pp. 129-134

Domingue, J. B., Eisenstadt, M, and Price, B. A. (1993) The VITAL Bug Location Methodology. VITAL Project Report DD231, The Open University, April, 1993.

Domingue, J. B., Price, B. A., and Eisenstadt, M. (1992) Viz: A Framework for Describing and Implementing Software Visualization Systems. In User-Centred Requirements for Software Engineering Environments (eds. D. Gilmore and R. Winder). Springer-Verlag.

Eisenstadt, M., Domingue, J., Rajan, T., and Motta, E. (1990) Visual Knowledge Engineering**.** IEEE Transactions on Software Engineering 16(10), pp. 1164-1177.

Ellis, C. A. and Gibbs, S. J. (1989) Concurrency Control in Groupware Systems. Proceedings of the ACM SIGMOD '89 Conference on the Management of Data Seattle, Washington.

Ellis, C. A., Gibbs, S. J., and Rein, G. L. (1991) Groupware: Some Issues and Experiences. Communications of the ACM 34(1), pp. 39-58.

Gaines, B. R. (1991) An Interactive Visual Language for Term Subsumption Languages. Proceedings of the 12th International Conference on Artificial Intelligence (IJCAI’91) Sydney, Australia, pp. 817-823.

Henderson, D. A. and Card, S. K. (1986) Rooms: The user of multiple virtual workspaces to reduce space contention in a window-based graphical user interface**.** ACM Transactions on Graphics 5(3), pp. 211-243.

Jonker, W., Kontio, J., and Motta, E. (1991) Definition and Positioning of the VITAL Project. VITAL Project Report ID732.1, PTT Research, Groningen, November 1991.

Jonker, W. and Motta, E. (1993) Functional design activity using $K_{BS}SF$ and VITAL-OCML. VITAL Project Report DD221 (Part III), PTT Research, Groningen, March 1993.
Kontio, J., and Rouge, A. (1991) VITAL Life Cycle Guide. VITAL Project Report VITAL/T1.1.1/DD111.
Koopman, M. R. J., Spee, J. W., Jonker, W., Montero, L., O'Hara, K., Mepham, M., and Motta, E. (1991) VITAL Conceptual Modelling, VITAL Project Report DD213, PTT Research, Groningen, December 1991.
Lapicque, J. Y., Lozinguez, Y., and Rouge A. (1991) The VITAL Verification and Validation Methodology. VITAL Project Report ID232.
Lenat, D.B. and Guha, R.V. (1990) Building Large Knowledge-Based Systems: Representation and Inference in the Cyc Project. Addison-Wesley.
Linster M. (ed.) (1992) Sisyphus '92: Models of problem solving. GMD report 630, GMD Sankt Augustin.
Motta, E., Stutt, A., O'Hara, K., Kuusela, J., Toivonen, H., Reichgelt H., Watt, S., Aitken, S., and Verbeck, F. (1991a) VITAL Knowledge Representation Language Specification, VITAL Project Report DD412, December 1991.
Motta, E., Rajan, T., Domingue, J., and Eisenstadt, M. (1991b) Methodological Foundations of KEATS, The Knowledge Engineers' Assistant. Knowledge Acquisition 3(1), pp. 21-47.
Motta, E., O'Hara, K., and Shadbolt, N. (1993) Grounding GDMs: A Structured Case Study. To appear in the Knowledge Acquisition Journal, Fall 1993.
Newell, A. (1982) The Knowledge Level. Journal of Artificial Intelligence 18.
O'Hara, K., Shadbolt, N., Laublet, P., Zacklad, M., and Leroux, B. (1992) Knowledge Acquisition Methodology, VITAL Project Report DD212, University of Nottingham, Nottingham, September 1992.
Open Software Foundation (1990) OSF/Motif Style Guide, revision 1.1 Open Software Foundation, 11 Cambridge Center, Cambridge, MA 02142, USA.
Schneiderman, B. (1987) Designing the User Interface: Strategies for Effective Human-Computer Interaction Addison-Wesley.
Steels, L. (1990) Components of Expertise. AI Magazine Summer Issue.
Steels, L. (1992) Reusability and Configuration of Applications by non-programmers. AI-Memo, 92-4, Free University of Bruxelles.
Stefik, M., Bobrow, D. G., Foster, G., Lanning, S., and Tatar, D. (1987) WYSIWIS Revised: Early Experiences with Multiuser Interfaces. ACM Transactions of Office Information Systems 5(2) pp. 147-167.
Tang, X. and Vuorenmaa, T. (1991) Requirement Specification in VITAL. VITAL Project Report BULL/T122/W/1.
Watt, S. N. K. (1993) Role Conflict in Groupware. Proceedings of the First International Conference on Intelligent and Cooperative Information Systems Rotterdam.
Watt, S. N. K., Domingue, J. B., and Motta, E. (1992) The Architecture of the VITAL Workbench. VITAL Project Report DD732a, The Open University, June 1992.
Wielinga, B. J., Schreiber, A. T., and Breuker, J. A. (1992) KADS: A Modelling Approach to Knowledge Engineering. Knowledge Acquisition Journal 4(1).