USING SOFTWARE VISUALIZATION TECHNOLOGY IN THE VALIDATION OF KNOWLEDGE BASED SYSTEMS[1]

John Domingue

Knowledge Media Institute,
The Open University, Walton Hall,
Milton Keynes, MK7 6AA.

J.B.Domingue@open.ac.uk

ABSTRACT

The validation of a Knowledge Based System (KBS) involves comparisons between an external reference model and a system's component parts. In this paper I describe how such comparisons can be aided by the application of software visualization technology. Software visualization is the use of filmcraft, cartoon animation and graphic design techniques to display data structures, programs, and algorithms. The described approach eases the task of mapping between the comparates by the use of dynamic code, problem solving architecture, and domain oriented visualizations of KBS execution.

1. INTRODUCTION

Software Visualization (SV) [Price et al., 1993] is the use of filmcraft, cartoon animation and graphic design techniques to display data structures, programs, and algorithms. Software visualization is basically the unification of algorithm animation and program visualization, two concurrent developments from the 1980s. Algorithm animations are high level characterisations of how data is manipulated during a program execution. Program visualization systems display graphical representations that are more tightly coupled with a program's code or data and show more or less faithful representations of the code as it is executing.

The validation of a Knowledge Based System (KBS) involves the comparison of a model residing within the KBS against a reference model which may be external to the system. A variety of techniques exist for carrying out the comparisons. The techniques are either carried out by hand or involve the use of a 'black box' program. In this paper I describe how software visualization technology could be used to support the validation process. In particular I address the following questions:

- what types of validation can SV aid?
- what validation techniques can most benefit from SV?
- what sorts of validation questions can be answered using SV?

I will also describe how SV was used in the ESPRIT-II funded VITAL[2,3] project [Shadbolt et al., 1994] [Domingue et al., 1993] to validate the Sisyphus I [Linster, 1992] application.

---

1 Accepted for publication at the ninth Knowledge Acquisition for Knowledge-Based Systems Workshop, Banff Canada, February 26-March 3, 1995.

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

3 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 author and not necessarily those of the consortium.

The rest of this paper is structured as follows. In the next section I will review some related SV work including KBS visualization systems. Section 3 contains an overview of KBS validation structured according to the three questions posed above. The following section outlines the SV supported approach to validation adopted in the VITAL project. Section 5 contains the example validation. Section 6 discusses the approach in detail, and the final section outlines some future work and conclusions.

2. SOFTWARE VISUALIZATION

2.1. Algorithm Animation

As may seem self evident algorithm animations are designed to explain algorithms. The explanations are generally used to educate students or aid researchers understand/improve/create algorithms. The birth of the algorithm animation field can be traced back to the Sorting Out Sorting (SOS) film [Baecker, 1981]. The film, which depicts various sorting algorithms, was developed for an undergraduate computer science course. After SOS, the next major contribution to the field was provided by the BALSA system [Brown, 1988]. BALSA allowed developers to create graphical views which could be tied into computer programs. As the programs ran, high level animations of the data would be displayed enabling aspects of the underlying algorithm to be viewed. BALSA was successfully used at Brown university to teach undergraduate computer science courses. A screen snapshot of BALSA is shown below.

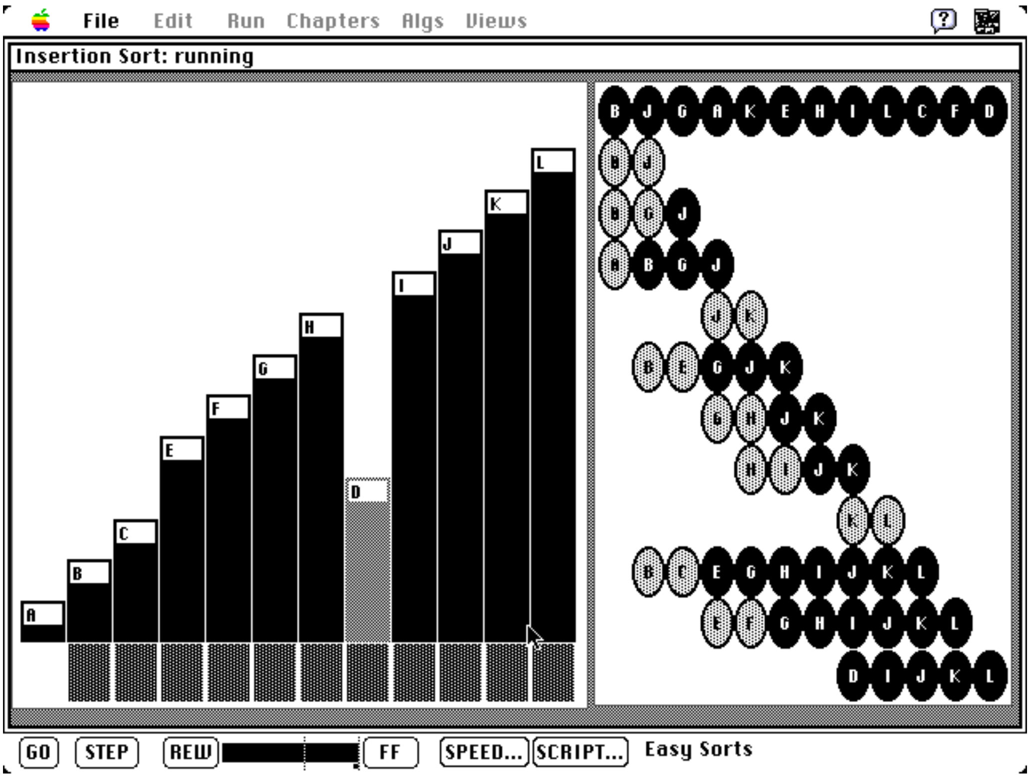

Figure 1. A screen snapshot from the Balsa system showing an animation of the insertion sort algorithm.

Since BALSA a number of systems have been created which have added smooth animation (TANGO [Stasko, 1990]), three dimensional graphics (POLKA [Stasko and Wehrli, 1993] and ZEUS [Brown and Najork, 1993]) and sound (ZEUS [Brown and Hershberger, 1991]). Figure 1 above shows a screen snapshot from the BALSA system animating an insertion sort algorithm. The window on the left represents each item to be sorted as a stick. The position of the item within the array is represented by its x co-ordinate and the item's value is represented by its height. The item currently being examined is shown in grey. The first row within the window on the right shows the initial array. Each subsequent row shows the history of each item, so, for example, the sixth row indicates that item E was compared with items K, J, G and B. E was smaller than the first three but larger than the last.

2.2. Program Visualization Systems

Program visualization systems use graphics to show control flow in program execution. Program visualization systems are broader in scope than algorithm animations since they provide a graphical representation of any program within a given language. Partly due to their broad scope and partly due to the fact that they are designed to show the execution of a program rather than an abstract algorithm, program visualization systems tend to use simpler graphics than algorithm animation systems.

In the rest of this section I shall review some KBS visualization systems.

2.2.1 Visualizing the Execution of KBS. Guidon-Watch [Richer and Clancey, 1985] is a graphic interface to NEOMYCIN, providing multiple views on a knowledge base. The knowledge base includes findings, hypotheses, rules, tasks and relations. The taxonomy of diseases is displayed graphically as a tree. Program visualization is provided by highlighting the diseases that are currently being considered. Diseases which were once considered but are no longer, are shown as dim boxes. Guidon-Watch also allows the Knowledge Engineer (KE) to view facts which have been inferred from the rulebase and the current stack of goals.

DTree [Mott and Brooke, 1987] uses a tree based representation to display rule structure. The trees are made up of nodes, each of which contain a feature and a colour. Features are global predicates which may be true or false. The value of a feature is indicated by its colour.

MRE (Multiple Representation Environment) [Brayshaw, 1994] is a PARLOG (a parallel extension to Prolog) debugging system based on the concept that understanding the execution of a program is a task of manipulating and managing an information space (the execution trace of the program). Three methods of searching the information space for bugs are included. Each method consists of a set of agents, where each agent type has its own searching technique.

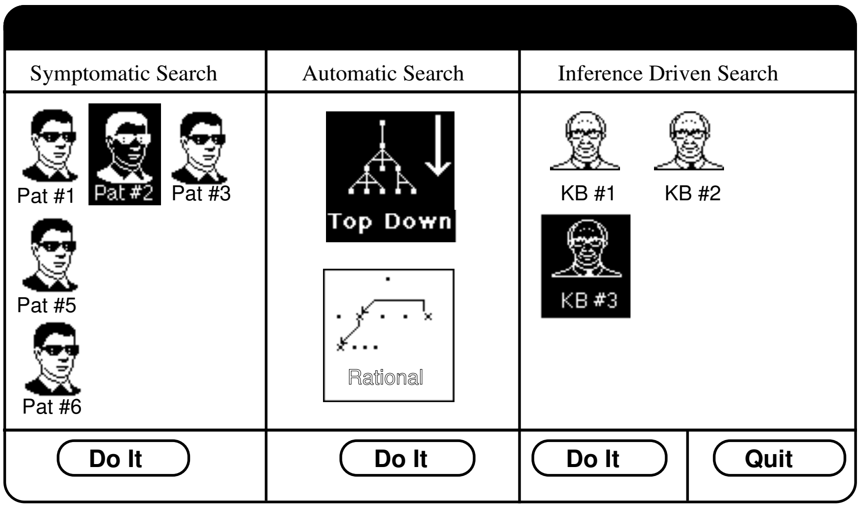

Figure 2. The debugging agent control panel in MRE.

KEATS [Eisenstadt et al., 1990] provided dynamic visualizations of a forward and backward chaining inference system and of a truth maintenance system. The forward chaining visualization system, called TRI (Transparent Rule Interpreter) [Domingue and Eisenstadt, 1989], used a 'musical score' metaphor to show coarse-grained behaviour. A simple example is shown in figure 3. Facilities were provided for the user to 'replay' execution, and to focus on particular moments of time and then to examine a fine-grained view to answer particular questions.

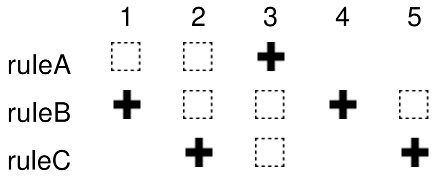

Figure 3. A TRI graph showing the execution of the rules ruleA, ruleB and ruleC over five cycles. The + means that a rule fired, a box means that a rule entered the conflict resolution set but did not fire. The rules fired in the order: ruleB, ruleC, ruleA, ruleB, ruleC. RuleA entered the conflict resolution set during the first 3 cycles only.

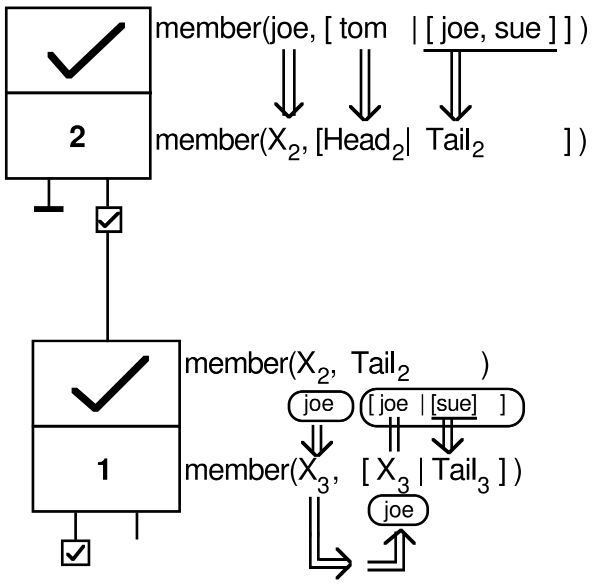

Figure 4. A TPM fine-grained view corresponding to solution of the query ?-member(joe, [tom, joe, sue]). The top (root) box indicates that the original goal matched with the head of the second member clause. The bottom (child) box indicates the subgoal member(joe [joe, sue]) matched with the head of the first member clause. Taken from [Eisenstadt and Brayshaw, 1988]. Reprinted by permission of Kluwer Academic Publishers.

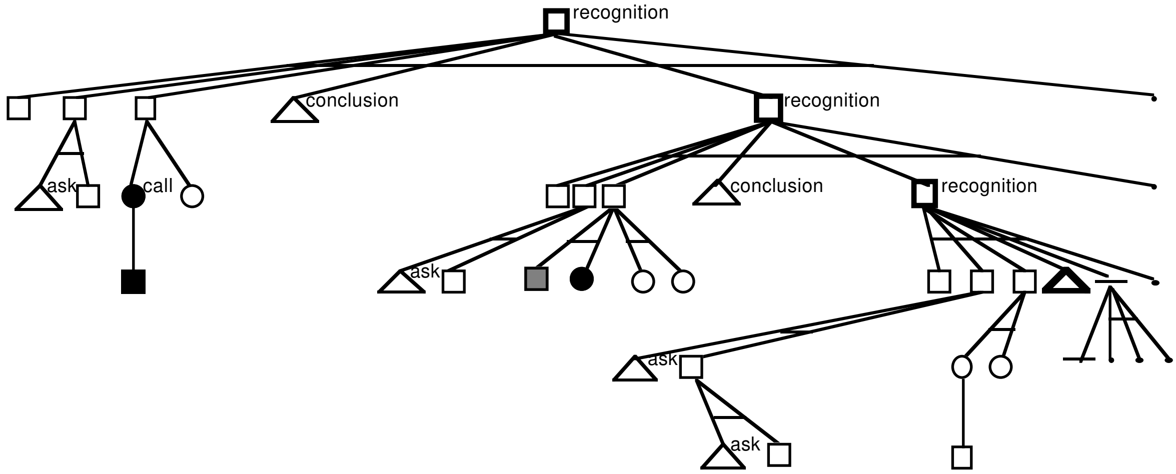

Figure 5. A coarse grained view in the middle of (re-) execution replay. Only certain nodes are labelled, at the user's discretion. Triangular nodes are 'compressed' sub-trees, which may be expanded upon request. White = success, black = failure; grey = succeeded earlier but failed on backtracking; very thick line = currently pending goal. The dashes and dots (far right and side) correspond to user-defined procedures and system primitives, respectively, which have not yet been executed at the moment depicted by this snapshot. Taken from [Eisenstadt and Brayshaw, 1988]. Reprinted by permission of Kluwer Academic Publishers.

The Transparent Prolog Machine (TPM) [Eisenstadt and Brayshaw, 1988] provides synchronised fine and coarse-grained visualizations (examples are shown in the two figures above) for Prolog based on augmented 'and/or' trees. TPM can visualize the execution of arbitrarily sized Prolog programs using several navigation techniques. For example, a programmer can specify a set of predicates whose subtrees should be collapsed (this is shown in figure 5). This enables irrelevant submodules of a program to be 'black boxed away'.

2.3. Design Level Visualization

Adhami et al. [1988] describe a graphical system able to display animated walk-throughs of traditional software engineering design documents. The system (and design documents) are not linked to the code, so the task of mapping between code behaviour and the design is left to the programmer.

3. KBS VALIDATION

KBS validation can be defined as [Benbasat and Dhaliwal, 1988]:

> "... the degree of homomorphism between the representation system, i.e., KBS, and the system that it is supposed to represent, i.e., the expertise source (Vandierendock, 1975)." p. 1-2

As SV can be used to provide explanations of the behaviour of a KBS from varying perspectives, SV can thus aid when an understanding of the behaviour of a KBS is required within a validation process. Figure 6 below shows the validation based life-cycle of KBS development contained in [Benbasat and Dhaliwal, 1988]. I shall now briefly define the terms in figure 6. The conceptual knowledge base is a skeletal structure created using preliminary elicitation techniques. Within KBS development the conceptual knowledge base is fleshed out into an elicited knowledge-base. A running system, the implementation, is constructed from the elicited knowledge base. Conceptual validation attempts to ensure that the assumptions and theories underlying the conceptual model are correct. Elicitation validation concentrates on the completeness and correctness of the instantiated conceptual model. Implementation validation, usually called verification, aims to ensure that the translation of the elicited knowledge base into a running system does not introduce any errors.
A KBS implementation can be validated by comparison with a source expert (either the original source or some other source). Functional evaluation explores performance quality, utility and

usability characteristics. The quality of the knowledge base and the advice given is evaluated by experts, whereas the usefulness (i.e. what are the benefits) and usability (i.e. ease of use and ease of understanding output) is evaluated by end-users. Representational validation examines the knowledge representation, processing logic and output.

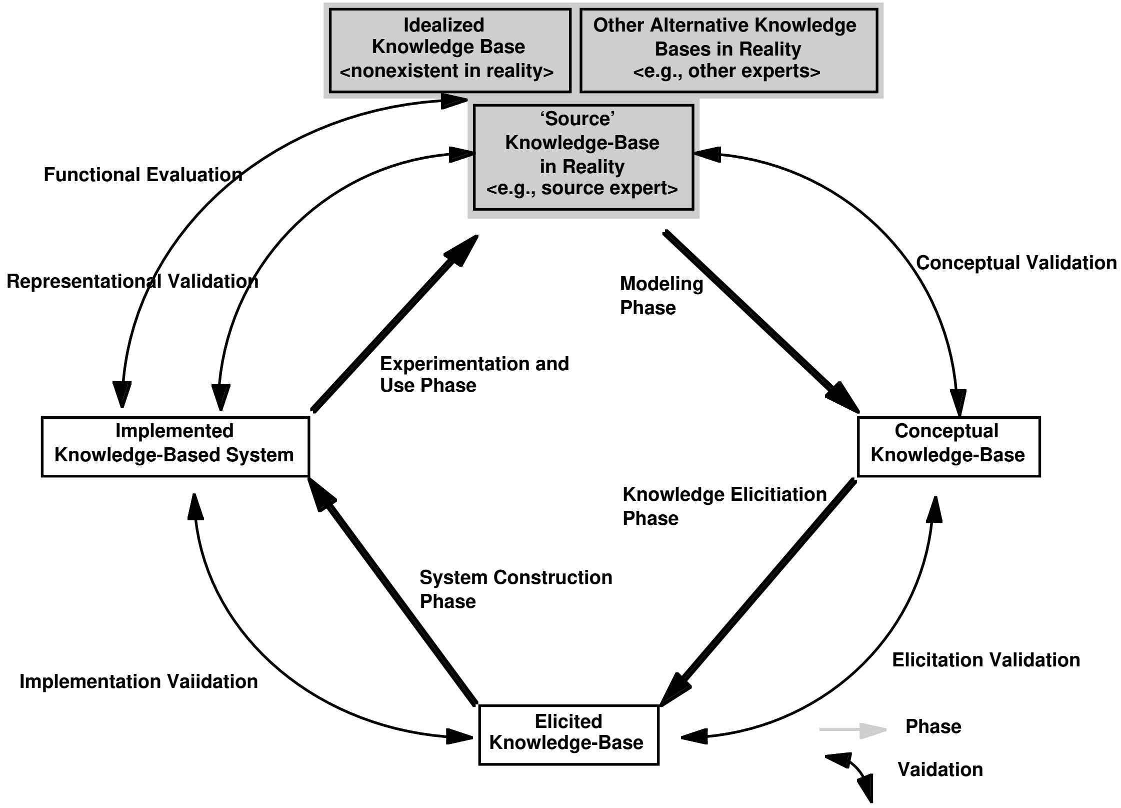

Figure 6. An overview of the KBS development life-cycle from a validation point of view (from [Benbasat and Dhaliwal, 1988] p. 1-4).

Obviously, SV can aid most in the types of validation which are linked with the KB's implementation. In section 4 I shall describe exactly how SV has been added to the above validation based life-cycle within VITAL.

The attributes of validation which determine the amount of aid SV can provide include the type of expertise source, the techniques used and the type of information required. I will now discuss each of these validation attributes in turn in the next three sections.

3.1. Types of Expertise Source

Gaines [1989] identified the following types of validation:

- objective - comparing to empirical data,
- pragmatic - used in applications,
- foundational - compared to other disciplines - 'deep knowledge',
- referential - comprehensibility of explanations in relevant reference community,
- subjective performance - compared to expert behaviour in applications,
- subject knowledge - compared to expert rationale in communication with reference community.

SV can aid most in objective and subjective performance validations. Comparisons in an objective validation may be facilitated if empirical data can be visualized. Within subjective performance validation visualizations can be constructed which use icons denoting domain objects and have domain specific behaviours.

3.2. Validation Techniques

Using SV in validation is not a new idea. O'Keefe et al., [1987] described a set of qualitative validation techniques which could be aided by visualization:

- Face Validation - project team members, potential users and domain experts compare system performance against human performance,
- Turing Tests - inspired by [Turing, 1950] the tests are as above but carried out blind, the evaluators do not know if they are evaluating a human expert or a KBS. Turing tests prevent evaluators biasing against a KBS, as was found to be the case in the evaluations of MYCIN [Gaschnig et al., 1983].
- Subsystem Validation - subsystems are validated individually. The advantages of this approach are:
  i) validation is carried out along with development,
  ii) subsystems are less complex and therefore easier to validate,
  iii) error detection is much easier since errors are localised,
  the disadvantages are:
  i) it may not be possible to observe the input-output behaviour of a subsystem,
  ii) successfully validating each subsystem does not imply overall system validity.
- Sensitivity Analysis - systematically changing system input, variable values and parameters over some range of interest and observing the effect on system performance.

3.2.1 Consistency Checking. There exist a number of systems which are able check the consistency of a KBS, e.g., CHECK [Nguyen et al., 1985], [Perkins et al., 1989]; Hybabs [Evertsz and Motta, 1991]; COVER [Preece, 1993] and COCO [Loiseau, 1994]. An excellent review of these types of systems can be found in [Preece et al., 1992]. Preece et al. categorise the types of inconsistencies (which they term anomalies) found by consistency checkers into the the hiearchy depicted below:

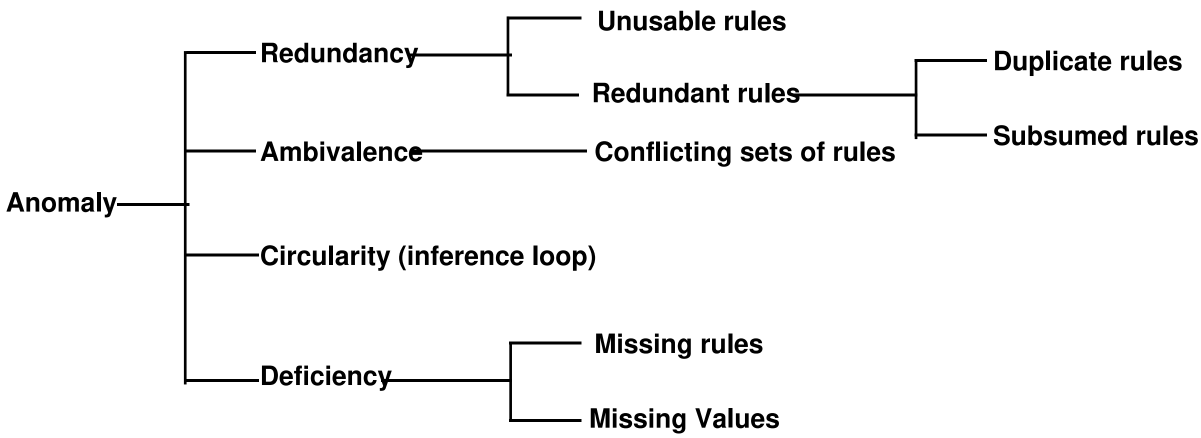

Figure 7. A classification of the types of inconsistencies found by consistency checkers (from [Preece et al., 1992 p. 117]).

I do not have space here to describe every node in the above tree here but I will instead give an example of each major type.

- redundancy - rules with equivalent antecedents and one or more equivalent consequent clauses,

- ambivalence - rules with equivalent antecedents but one or more consequent clauses are contradictory,
- circularity - a set of cyclic rules.
- difficiency - some of the rules required to produce a certain solution are missing.

The information gathered by such systems is typically used to generate static output often in the form of canned text. This information could be used to augment SV. For example, individual rules firings displayed in a TRI style diagram (see figure 3) could be colour coded according to the above categories of inconsistencies.

I shall elaborate on how SV can aid in the above validation techniques in section 6.2.

3.3. Types of Validation Question

Neches et al., [1985] in the development of the Explainable Expert System (EES) identified four types of validation questions:

- Questions of justification - information about the purpose underlying some aspect of the system, e.g. why is the system concerned with parameter/goal/action,
- Questions of timing or appropriateness - information about the course of a KBS reasoning process e.g. when did the system consider/reject goal/action/conclusion,
- Questions of definition or function - the meaning of a particular concept used within the system e.g. what does this term mean?
- Questions of capabilities - information on the abilities and limitations of a system e.g. what methods does the system use/avoid in achieving its goals.

I shall describe exactly how SV tools can help in answering the above types of questions in section 6.3.

4. USING SV TECHNOLOGY IN KBS VALIDATION IN VITAL

The design of a KBS in VITAL uses a knowledge-level modelling language developed within the project called OCML [Motta, 1995]. OCML models can be created in a plain text editor or using a graphical interface. As the visualizations use the graphical OCML I shall concentrate on the graphical OCML for the rest of the paper. An OCML model contains descriptions of the problem solving architecture (PSA) and the domain-level knowledge structures. Domain models comprise the concepts, instances, relations and relation instances in the domain. Problem solving architecture models consist of tasks and roles connected by data flow and control flow relations representing the control and data flow of a design. A PSA can be thought of as an instantiated set of problem solving methods. Rules can be used to represent complex relations and to specify how tasks are carried out. Once a design has been sketched it can be 'operationalised' by linking terminal nodes to an implementation layer using a 'represented as' node property and compiling the model. Once a model has been compiled, a task can be selected from the model and executed.

The problem for a KE, in carrying out validation, is to map between a source expert and the various models within a KBS. Within VITAL, the approach has been to reduce the mapping effort required by extensive use of SV techniques providing synchronised design, domain and code views onto a KBS. The use of SV in VITAL is summarised in the diagram below, adapted from figure 6.

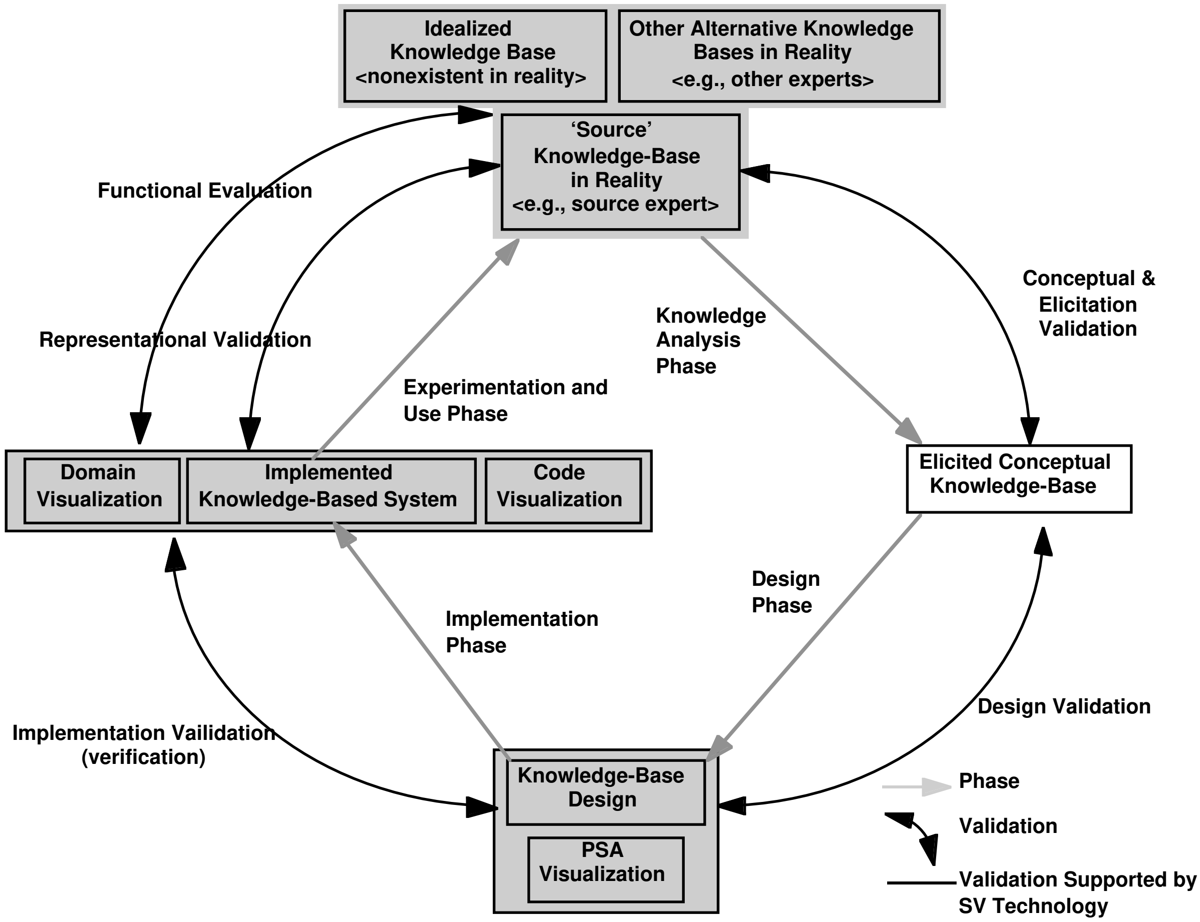

Figure 8. A summary of SV use in KBS evaluation within VITAL.

I shall now explain figure 8. In the VITAL methodology [Jonker et al., 1991] we distinguish between a fleshed out conceptual model and a 'system oriented' design model. The skeletal and elicited conceptual model are considered to be a (growing) single entity.

We can see that functional evaluation and representational validation now use domain and code visualizations. The code visualization allows logical processes and the interactions between the knowledge representation components to be examined. The domain visualization provides a domain oriented view of the execution. The two visualizations act as stepping stones between the implementation and the source expertise. The domain and code visualizations are also used in implementation validation, in conjunction with the PSA visualization. A PSA visualization contains a graphical explanation of the interactions between the tasks during an execution. Mismatches between the code, domain and PSA visualizations can indicate the location of possible design to implementation translation errors.

The following section illustrates some of the features of the approach using an example application.

5. THE VALIDATION OF SISYPHUS I

5.1. Sisyphus I The Room Allocation Problem

The Sisyphus I problem [Linster, 1992] is one of the standard problems used within the knowledge acquisition community to compare KBS systems and methodologies. The problem is to assign a set of researchers from a laboratory to a set of rooms within a new building. Various constraints are placed on the allocation of rooms. The constraints are:

i) the head of the group should be in a central office,

ii) the secretaries should be close to the head of the group,

iii) the project managers should have maximum access to the secretaries and the head of the group,

iv) smokers and non-smokers should not share an office,

v) as far as possible researchers sharing an office should be working on different projects.

The implementation of this example is described in detail in [Motta et al, 1994]. I shall now briefly outline how the Sisyphus I application was created. In the rest of section 5 I shall describe how the visualizations were used in the validation process.

5.2. The Sisyphus I Design in VITAL

The Sisyphus I design in VITAL consists of three modules. The first module contains the concepts and instances such as the researchers and rooms. The second contains the relations such as ‘being within a room’, and a final module contains the problem solving architecture. The PSA module is now described as it is used within the PSA visualization.

Figure 9 shows the control and data flow for the Sisyphus I application. Tasks are represented by ovals and rôles by rectangles (note that I distinguish between OCML rôles and Sisyphus roles with a circumflex). Tasks specify knowledge level activities. A rôle represents some segment of task input or output data (defined in a concept/instance or relation module). The cardinality of a rôle is indicated by a numeral in the top right corner of the rectangle. The solid lines in the diagram are control flow links and the dashed lines are data flow links. Duplications or aliases (two nodes representing the same task or role) are indicated by the use of an italic font. PSA diagrams can also contain control constructs. Three are present in the figure above. The hexagonal node is a choice control construct. In a choice construct control flow passes to one of the subtasks depending on the value of the input role. The rounded rectangle labelled ‘Return1’ is a return construct. The execution of a return construct passes control back to the root task (shown as a double oval). The remaining rounded rectangle is a sequence control construct. A sequence construct is used for aesthetic purposes, to make diagrams grow vertically rather than horizontally.

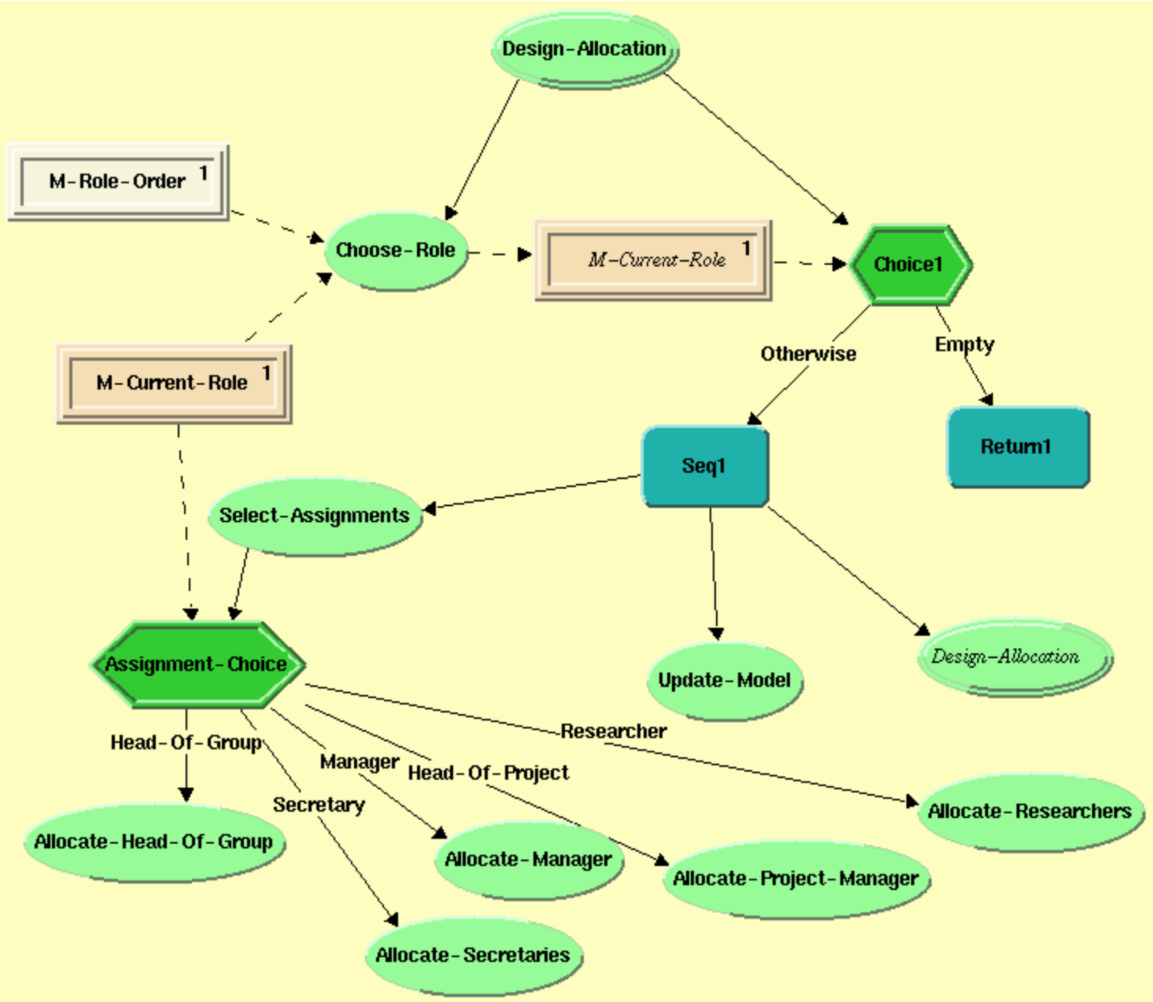

Figure 9. The Sisyphus I problem solving architecture.

I shall now explain the design specified by the diagram. In order to carry out the design allocation task we choose a role using the role-order and the current-role. The new current-role determines which assignment task (allocate-head-of-group, allocate-secretaries etc.) is chosen. We keep doing this until the current-role becomes empty. The rôles current-role and role-order are defined in the relations module.

5.3. Domain Oriented and PSA Visualizations

The visualizations are designed to be used post-mortem, although a capability for live visualization is provided. Each visualization can be replayed using a replay panel having a functionality similar to a tape or video recorder. The visualizations are synchronised so that replaying one replays the others. The KE can specify what type of visualization is used at the domain level and the types of visualization used at the code level (on a task by task basis if required). For Sisyphus I, a specially constructed room-person domain visualization and a predefined TRI style visualization were used. Figure 10 below shows the domain oriented visualization at the end of replay.

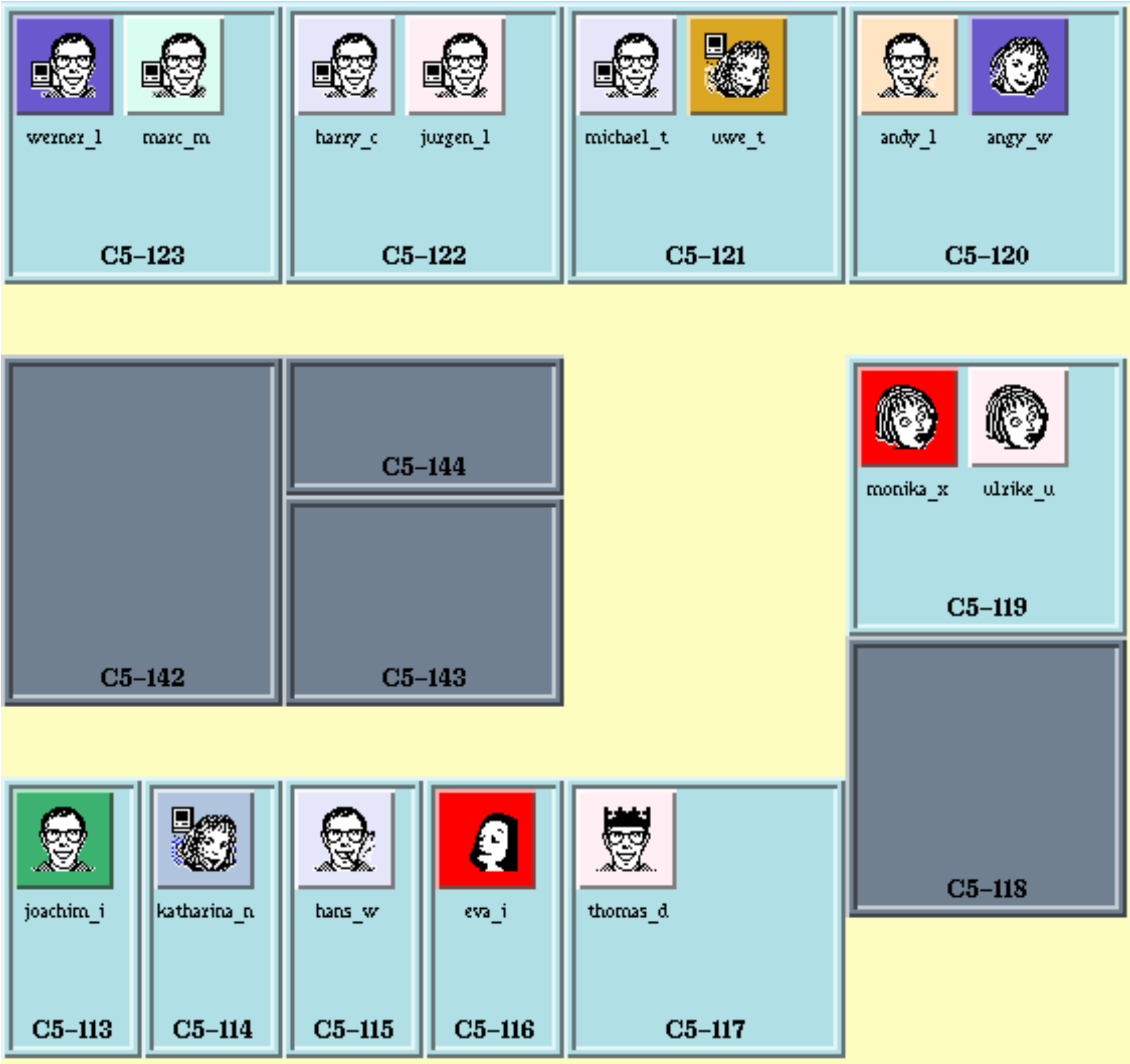

Figure 10. The Sisyphus I domain oriented visualization at the end of execution.

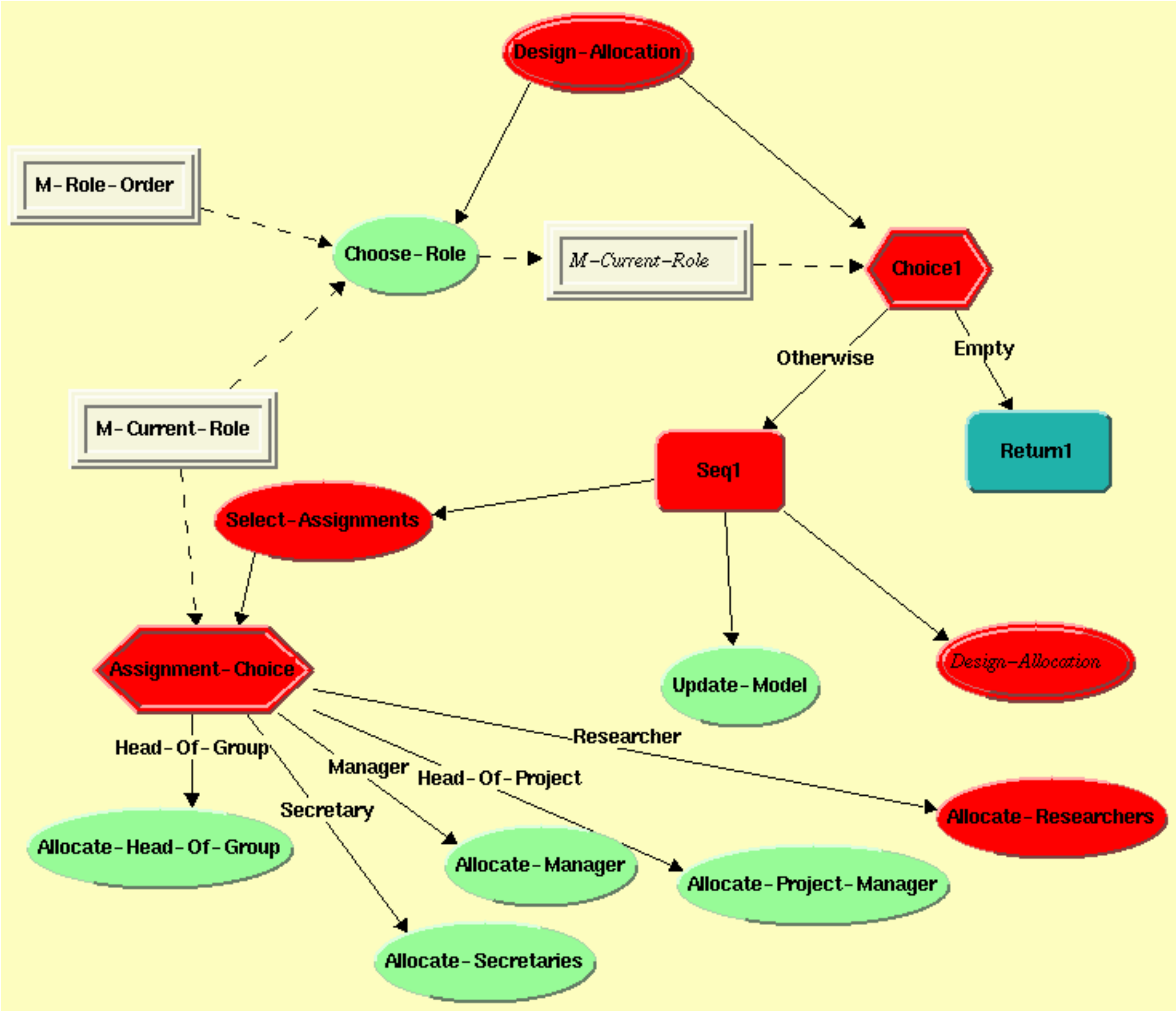

Figure 11. The PSA diagram after replaying to the paused point.

Within the visualization shown in figure 10 each icon represents a member of the group. The colour of each icon indicates the current project of the member. The visual appearance of an icon indicates a person's role within the group in the following way:

-  or  - researcher,
-  or  - smoker,
-  or  - hacker,
-  or  - hacker and smoker,

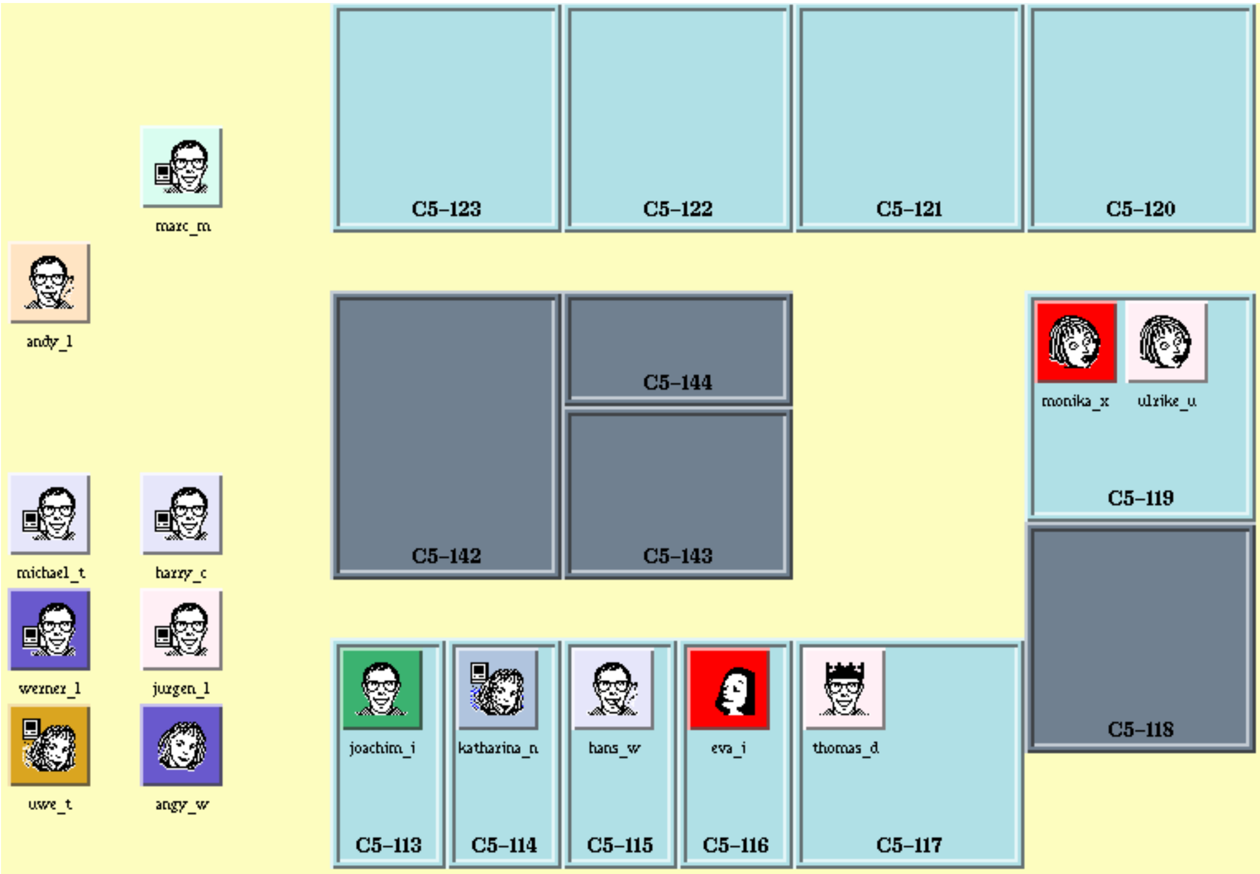

Figure 12. The domain level visualization just before allocate-researchers is executed.

Figure 13. A code level view of allocate-researchers task.

-  - manager,
-  - secretary,
-  - head of group.

As the visualization runs the icons slide into the rooms. We can see from the visualization in figure 10 that the fourth constraint has been violated. The rooms C5-120 and C5-121 contain a smoker and non-smoker. Notice that a mismatch between expected behaviour and the KBS behaviour is extremely easy to notice using the domain visualization. The KE now needs to find the part of the executable code and problem solving architecture responsible for the incorrect behaviour.

The KE adds a pause point, for the 'assign to room' event, to the angy_w icon. This is achieved by selecting the angy_w icon and choosing 'Pause' from the 'Operations' menu. The KE then replays the visualization from the beginning of the execution from the PSA document. Figure 11 above shows the PSA diagram when the replay halts. Figure 12 shows the domain visualization just before the final task allocate-researchers is executed. After the allocate-researchers task is executed the domain visualization has the same appearance as figure 10. PSA visualizations are created by highlighting in red (a dark grey in greyscale) the currently executing tasks and the accesses to rôles. When the pause point is reached a dialogue window is popped up (not shown) and replay continues until the end of the current task. We can see from figures 10, 11, and 12 that in the execution of the allocate-researchers task, eight researchers were allocated to rooms C5-120 to C5-123. As the visualization runs we see the researchers fill the rooms in numeric order (C5-120 then C5-121 then C5-122 etc.). We can now examine the execution of the allocate-researchers task at the executable code level. This is achieved by selecting the task and choosing 'Show Code Level Visualization' from the 'Operations' menu.

5.4. Code Level Visualizations

Figure 13 above shows the execution history for the allocate-researchers task using a TRI style visualization described in section 2.2. We can see from the figure that only one rule took part in the execution using working memory patterns belonging to the 'in-room', 'room' and 'researcher' predicates. As this task is represented by four rules the execution seems suspect. We can also see that the rule fired four times, presumably allocating a single room on each firing. We can view the first rule firing (corresponding to the allocation of andy_l and angy_w to room C5-120) in greater detail by selecting the first cross and choosing 'Show Fine Grained View' from the 'Operations' menu. Figure 14 below shows a portion of the fine grained view for the first firing.

A fine grained view for a rule firing contains the rule definition followed by all the instantiations current for the rule. Instantiations eliminated by a conflict resolution strategy are shown in a plain font with the name of the eliminating strategy. The firing instantiation is displayed first, using an italic font. If a cycle number in the table view (figure 13) is selected, or the cycles are 'played through' using a replay panel, the fine-grained view updates itself to display information appropriate to the current cycle. We can see that the firing instantiation allocates andy_l and angy_w to room C5-120 even though andy_l's smoker property has value of t and angy_w's smoker property has a value of nil. Looking at the definition of the rule we can see that the smoker property is bound to ?v for researcher ?r1 and ?u for ?r2. The property should be bound to the same variable. We can now fix this bug by editing the source code.

```
(def-rule researcher-rule-1 if
 ((researcher ?r1 allocated? nil group ? works-with ? hacker ?t smoker ?v project ?p1))
 ((researcher ?r2 allocated? nil group ? works-with ? hacker ?t smoker ?u project ?p2))
 (not-eri-equal ?r1 ?r2)
 (not-eri-equal ?p1 ?p2)
 ((room ?room allocated? nil usable? t next-to ? central? ? size 2))
 (then)
 (list (format t ~3%~s and ~s go in room ~s~3% ?r1 ?r2 ?room) (note-eri-allocation ?room ?r1) (note-eri-allocation ?room ?r2))
 ((in-room ?r1 ?room))
 ((in-room ?r2 ?room))
 (modify-concept-pattern (quote room) ?room (list (quote allocated?) (quote t)))
 (modify-concept-pattern (quote researcher) ?r1 (list (quote allocated?) (quote t)))
 (modify-concept-pattern (quote researcher) ?r2 (list (quote allocated?) (quote t)))
cycle: 1

(researcher andy_l allocated? nil group unbound works-with nil hacker nil smoker t project tutor2000)
(researcher angy_w allocated? nil group unbound works-with (marc_m werner_l) hacker nil smoker nil project respect)
(not-eri-equal (elt-no-error (elt-no-error lhs 0) 1) (elt-no-error (elt-no-error lhs 1) 1))
(not-eri-equal (elt-no-error (elt-no-error lhs 0) 13) (elt-no-error (elt-no-error lhs 1) 13))
(room c5-120 allocated? nil usable? t next-to (c5-121 c5-119) central? nil size 2)
-->
(list nil 9 10)
(in-room andy_l c5-120)
(in-room angy_w c5-120)
(modify-concept-pattern room c5-120 (allocated? t))
(modify-concept-pattern researcher andy_l (allocated? t))
(modify-concept-pattern researcher angy_w (allocated? t))


(researcher werner_l allocated? nil group unbound works-with (angi_w marc_m) hacker t smoker nil project respect)
(researcher werner_l allocated? nil group unbound works-with (angi_w marc_m) hacker t smoker nil project respect)
(not-eri-equal (elt-no-error (elt-no-error lhs 0) 1) (elt-no-error (elt-no-error lhs 1) 1))
(not-eri-equal (elt-no-error (elt-no-error lhs 0) 13) (elt-no-error (elt-no-error lhs 1) 13))
(room c5-123 allocated? nil usable? t next-to (c5-122) central? nil size 2)
-->
eliminated by not-first
```

Figure 14. A fine grained view of the rule researcher-rule-1.

5.5. The Construction of Visualizations

Unlike PSA and code visualizations, domain visualizations represent ‘real world behaviours’ and are therefore idiosyncratic. This means that it is not possible to provide a library of predefined visualizations, and each domain visualization therefore must be constructed. To facilitate the construction of ‘one off’ domain visualizations VITAL provides a framework [Domingue et al., 1992] and programming language for defining SV systems. The framework, called Viz, has the following components:

- *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.

A visualization can be defined using the metaphor of players, states, events, views and mappings. Once defined the visualization is attached to the appropriate parts of the implementation layer using the PSA diagrams. Using a PSA document we can view the code level representation for a task using a special editor. Within the editor it is possible to highlight segments of the code which will be used to create the domain oriented execution history. The figure below shows the segment of the code level representation for the manager assignment task highlighted for the Sisyphus I domain view.

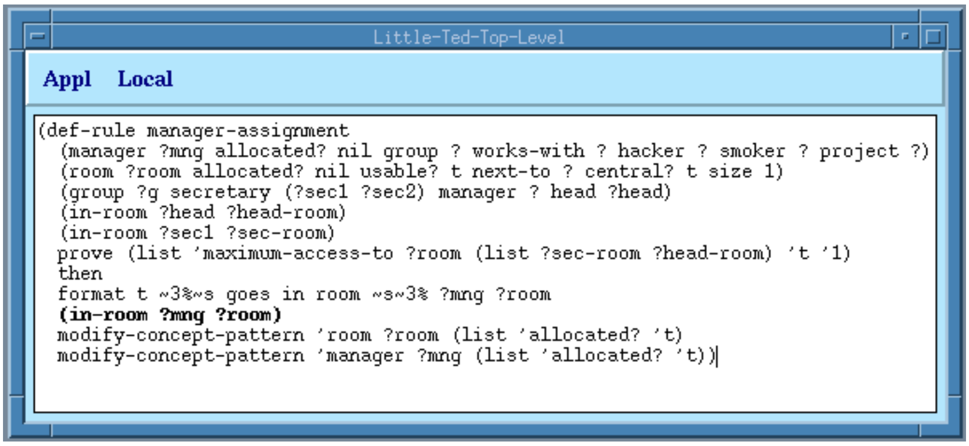

Figure 15. An annotated Sisyphus I rule for the domain visualization.

Within VITAL Viz has been used to create visualizations for two forward chaining rule based systems, a commercial Prolog system, the VITAL knowledge representation framework [Gaspari and Motta, 1994], the VITAL task design interface, a commercial Lisp system, and Sisyphus I and II [Yost, 1992].

6. DISCUSSION

6.1. Visualization Support Provided

In the example we saw how a domain visualization enabled us to see quickly a mismatch between a specified behaviour (the list of constraints) and a system's behaviour. Locating the part of the code responsible for this was aided by:

- the fact that the visualizations are synchronised,
- the ability to add pause points to the domain view,
- the ability to look at a 'task slice' of the code level execution, and
- the ability to view the code executions at coarse and fine-grained views.

A summary of the support the workbench provides in design and validation is depicted in figure 16 below.

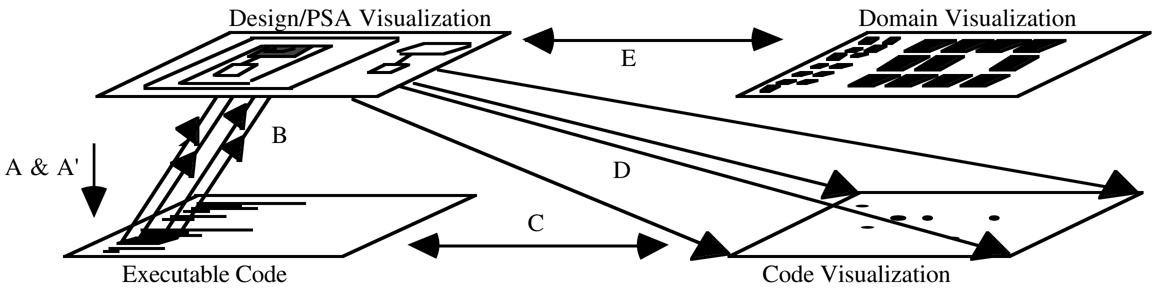

Figure 16. The visualization support provided by VITAL.

The labels in figure 16 above denote the following:

A the transformation, carried out by the OCML compiler, of the design model into executable code,

A' the ability to view or edit the source code representation for a design model component,

B the ability to map back from segments of code to the associated design model component,

C the bi-directional mapping between the static code representation and the code visualizations,

D the ability to obtain a code level visualization for a particular execution of a design model fragment,

E the synchronisation of the PSA and domain visualizations.

6.2. How SV can Aid Certain Validation Techniques

6.2.1. Face Validation. We can see from the example that with SV technology face validation becomes relatively easy. Simple invalid behaviours such as a researcher moving to one room then to another, or a particular role not being allocated become obvious. Some quite tricky concepts become explainable. For example, from the expert protocol it is not obvious what the concept of 'maximum access to' means in the third constraint. I would argue that the definition used in the design model, which uses four rules, would be harder for a domain expert to understand than just observing the domain visualization. It should be noted here that the creation of a domain level visualization is easier for a KBS than for a traditional program because domain knowledge is expressed explicitly and separately. This fact is exploited in the support provided for Turing Tests and is described in the next section.

6.2.2 Turing Tests. Enabling domain experts and KBSs to compete 'on a level playing field' in a Turing test is a non-trivial problem. Gaschnig et al. [1983] described several problems:

> "Since the typescript protocol generated by an expert system cannot very well be disguised to resemble human dialogue, the desire to blind a study limits the amount of evidence that can be made available to the panel of expert judges..." p. 251.
>
> "The experts engaged in evaluating system performance must have full access to all aspects of behaviour, so that they can push and probe, looking for weaknesses and deficiencies. This would seem to rule out blinded, comparative studies as an appropriate framework for expert-system evaluation..." p. 252.

Domain visualizations alleviate this by providing a rich description language. In conjunction with a KE the domain expert can create a scripted a visualization. I shall now describe how this is done. The KE creates a mapping between domain concepts and/or instances to graphical icons. This can be carried out during knowledge elicitation or at a later stage. An example mapping from Sisyphus I is shown below:

```
(RULE researcher-icon-map-5
  (researcher ?r1 smoker t hacker t)  &
  (female-name ?r1)
  THEN
```

```
)
```

Once a set of mappings have been created a specialised Sisyphus I script editor can be created. A snapshot of one in use is shown in figure 17 below.

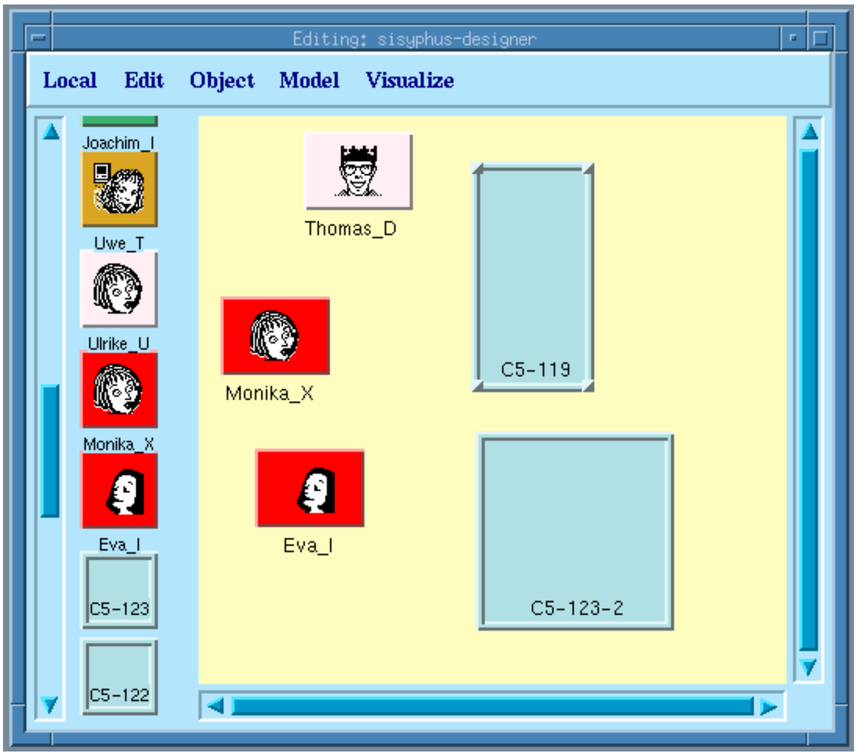

Figure 17. A snapshot of a Sisyphus I script editor. The palette on the left is a well containing all the YQT researchers and the rooms.

Using the above editor a scenario, with the appearance of figure 10, can be created by dragging the appropriate icons from the palette on the left of the window. The domain expert can then carry out a task, in this case a room allocation by direct manipulation. The manipulations can be recorded and replayed later on. An SV supported Turing test thus involves the comparison of a domain scripted visualization against a KBS domain visualization.

6.2.3. Subsystem Validation. SV can aid in subsystem validation. Using the design document we can fast forward to a particular task. If a task is executed more than once we can choose which execution to view in detail. Navigation facilities are also provided at the implementation level. This eliminates the first problem outlined by O'Keefe (see section 3.2), that of observing the input-output behaviour of a subsystem.

6.2.4. Sensitivity Analysis. Sensitivity analysis involves the comparison of separate KBS executions. In previous work [Domingue, 1988; 1990] I outlined how code level KBS executions could be compared. Two or more design visualizations can be compared by replaying the executions on the same design document and assigning a different highlighting colour to each one. The attached domain visualizations can then be viewed simultaneously (given adequate screen space).

6.2.5. SV and Consistency Checking. Visualizations can aid in inconsistency checking in two ways. Firstly, some simple inconsistencies, such as the same person being in two different rooms can be easily noticed. Secondly, conditional pause points (expressed in Lisp or Prolog) and the replay fast-forward can be used in conjunction to search for inconsistencies. This can be extended to cover verification by viewing or searching over two or more visualizations. By viewing we could spot errors such as:

- inappropriate design module use - for example, managers moved to rooms in the domain view while the design visualization indicated that the allocate secretaries task was executing,
- inappropriate code module use - for example, rules belonging to one role firing within a different role's task execution,
- incorrect design control - for example, a (correct) allocation task executing twice in succession instead of just once in a room allocation,
- incorrect code control - a (correct) rule firing too many times within a task execution.

Conditional pauses over two or more visualizations can be used to locate more subtle bugs such as 'was there any time during the execution when the current role within the domain, design or code level visualization were not all the same?'. Information gathered by inconsistency checking systems could be used to augment SV displays. Figure 18 below shows an example.

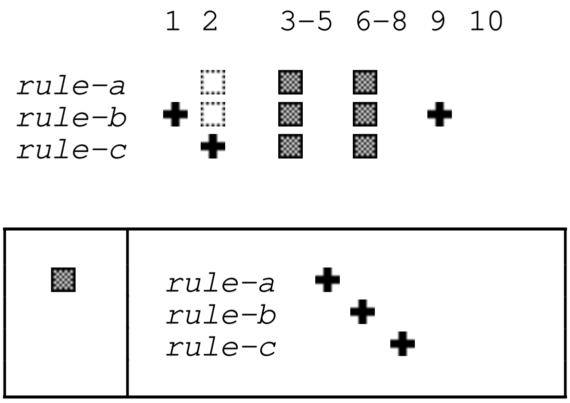

Figure 18. A forward chaining visualization augmented by information from a consistency checker.

In the figure above the partial cycle cliché (the three rules rule-a, rule-b, and rule-c firing in order) was spotted using information from a consistency checker. The information was used to collapse the number of columns in the visualization. Clichés like this could be used to highlight potential trouble spots or to compress irrelevant segments of the execution history.

6.3. How SV Aids in Answering Validation Questions

The use of SVs allows the four types of questions covered by the EES [Neches, 1985] (see section 3.3) to be answered in the following ways:

i) questions of justification - the purpose for system actions are explained from domain, problem solving architecture and code points of view by the visualizations,

ii) questions of timing or appropriateness - as well as viewing the synchronised visualizations we can conditionally pause on a specific event occurring to a particular object within a domain, problem solving architecture or code oriented view,

iii) questions of definition or function - the role of domain objects are self-evident from their appearance and behaviour,

iv) questions of capabilities - the capabilities of a system are displayed at the domain, problem solving architecture and code level.

7. FUTURE WORK AND CONCLUSIONS

I am curently integrating the SV validation approach described in this paper into the VITAL methodology. In particular I am focussing on how an expert scripted visualization (ESV) could aid in driving knowledge elicitation and analysis. One can envisage an expert sitting with the KE eliciting domain concepts and instances with appropriate mappings to icons. Once a scenario had been created the expert would then perform a test case using the purpose built environment. The expert manipulations would be mapped by the KE (using appropriate tools) into an ever growing PSA.

A dimension not covered in this paper is ‘how does SV based validation scale up?’ This is currently being explored in the context of the Sisyphus II problem [Yost, 1992]. Our current solution is to provide task slices of the domain visualization.

Within the knowledge acquisition field, validation is of importance because a KBS cannot be totally formally specified. Current environments provide static knowledge level visualizations with little or no link to code level execution. This places the KE in the arduous situation of having to synthesise knowledge and code level connections whilst burdened with the difficulties of the validation task. The approach outlined in this paper alleviates this by integrating SV techniques into the design and implementation tools and by providing static and dynamic:

• domain views which easily map onto the domain ontology,
• problem solving architecture views showing how components of the design interacted, and
• code views showing the execution at the code level.

KBS development is enhanced for the various involved communities in three ways. Firstly, systems builders benefit because problem areas can be identified more quickly. Secondly, communication between system builders, domain experts and end users are improved, thus increasing acceptability chances. Finally, researchers gain new insights into how a KBS reasons.

REFERENCES

Adhami, E., Shand, J., and McNeile, A.T. (1988). A Graphical Execution Environment for JSD Specifications. Proceedings of the Second British Conference on Software Engineering, Liverpool, England, pp. 138-142.

Baecker, R.M., and Sherman, D. (1981) Sorting Out Sorting. narrated colour videotape, 30 minutes, presented at ACM SIGGRAPH '81. Published by Morgan Kaufmann, Los Altos, CA.

Benbasat, I. and Dhaliwal, J. S. (1988) A Framework for the Validation of Knowledge Acquisition. Proceedings of the 3rd Knowledge Acquisition for Knowledge-Based Systems Workshop. Banff, Alberta, Canada. 6-11 November, 1988.

Brayshaw, M. (1994) Information Management and Visualization for Debugging Logic Programs. Ph.D. Thesis, Human Cognition Research Laboratory, The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK.

Brown, M.H. (1988) *Algorithm Animation.* ACM Distinguished Dissertations, MIT Press, New York.

Brown, M. H. (1991). Zeus: A System for Algorithm Animation and Multi-View Editing. In Proceedings of IEEE Workshop on Visual Languages, New York: IEEE Computer Society Press.

Brown, M. H. and Hershberger, J. (1991). Color and Sound in Algorithm Animation Technical Report No. 76a. DEC Systems Research Center, Palo Alto, CA.

Brown, M. H. and Najork, M. A. (1993). Animating Algorithms in 3D (Technical Report No. TBA). DEC Systems Research Centre, Palo Alto, CA.

Domingue, J. (1988) TRI: The Transparent Rule Interpreter. Proceedings of Expert Systems’88 in Shadbolt, N. (Ed.) *Research and Development in Expert Systems VII,* Cambridge University Press, December, 1988.

Domingue, J. and Eisenstadt, M. (1989) A New Metaphor for the Graphical Explanation of Forward Chaining Rule Execution. In Proceedings of The Twelfth International Joint Conference on Artificial Intelligence (Detroit, MI), pp. 129-134.

Domingue, J. (1990) Compressing and Comparing Metric Execution Spaces. In D. Diaper, D., Gilmore, G. Cockton, G. and B. Shackel (Eds.) Proceedings of the 3rd IFIP Conference on Human-Computer Interaction, Cambridge, *Human-Computer Interaction: Interact’90,* North-Holland, August, 1990.

Domingue, J., Price, B. and Eisenstadt, M. (1992) Viz: a framework for describing and implementing software visualization systems. In D. Gilmore and R. Winder (Eds.) *User-Centred Requirements for Software Engineering Environments,* Springer-Verlag.

Domingue, J., Motta, E., and Watt, S. (1993) The Emerging VITAL Workbench. In Aussenac, N., Boy, G., Gaines, B., Linster, M., Ganascia, J.-G., and Kodratoff, Y. (Eds.). *EKAW ‘93 Knowledge Acquisition for Knowledge-Based Systems.* Lecture Notes in Artificial Intelligence, LNCS 723, Springer-Verlag.

Domingue, J., Eisenstadt, M., and Price, B. (1994) The VITAL Visualization Tool Report. VITAL Project Report DD342. Human Cognition Research Laboratory, The Open University, Walton Hall, Milton Keynes, MK7 6AA, UK.

Eisenstadt, M. and Brayshaw, M. (1988) The Transparent Prolog Machine (TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4), pp. 277-342.

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

Evertsz, R., and Motta, E. (1991) The Abstract Interpretation of Hybrid Rule/Frame-based Systems. In E. Ardizzone, S. Gaglio, and F. Sorbello (Eds.) *Trends in Artificial Intelligence.* Lecture Notes in Artificial Intelligence, Springer Verlag.

Gaines, B. (1989) Integrating Issues in Knowledge Support Systems. *International Journal Of Man-Machine Studies,* 31(5) pp. 495-515.

Gaschnig, J. Klahr, P. Pople, H. Shortliffe, E. and Terry, A. (1983) Evaluation Of Expert Systems: Issues And Case Studies. In Hayes-Roth, F., Waterman, D. A., and Lenat D. B. (Eds.) *Building Expert Systems* Addison-Wesley, New York. pp. 241-280.

Gaspari, M., and Motta, E. (1994) Symbol-level Requirements for Agent-level Programming. Proceedings of the 11th European Conference on Artificial Intelligence, ECAI '94.

Guida, G., and Mauri, G. (1993). Evaluating Performance and Quality of Knowledge-Based Systems: Foundation and Methodology. *IEEE Transactions on Knowledge and Data Engineering.* Vol. 5. No. 2, April, 1993.

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.

Linster M. (Ed.) (1992) Sisyphus '92: Models of problem solving. GMD report 630, GMD Sankt Augustin.

Loiseau, S. (1994) A Method for Checking and Restoring the Consistency of Knowledge Bases. *International Journal of Human-Computer Studies,* 40(3) pp. 425-442.

Mott, P. and Brooke, S. (1987) A Graphical Inference Mechanism. *Expert Systems,* May, 2(4) pp. 106-117.

Motta, E., O'Hara, K., and Shadbolt, N. (1994) Grounding GDMs: A Structured Case Study. *International Journal of Human-Computer Studies,* 40, March, pp. 315-348.

Motta, E. (1995) KBS Modelling in OCML. Proceedings on the Workshop on Modelling Languages for KBS, Vrije Universiteit Amsterdam, January 3-31, 1995.

Neches, R., Swartout, W. R. and Moore, J. D. Enhanced Maintenance and Explanation of Expert Systems Through Explicit Models of Their Development. *IEEE Transactions on Software Engineering,* 11(11), pp. 1337-1351.

Nguyen, T. A., Perkins, W. A. Laffey, T. J. and Pecora D. (1985) Checking an Expert System's Knowledge Base for Consistency and Completeness. Proceedings of the Ninth International Joint Conference on Artificial Intelligence, 18-23 August, 1985, Los Angeles California, pp. 375-378.

O'Keefe, R., Balci, O. and Smith, E. P. Validating Expert System Performance. *IEEE Expert,* 4(11) Winter, 1987.

Perkins, W.A., Laffey, T.J., Pecora, D. and Nguyen, T.A. (1989) Knowledge Base Verification In Guida, G. and Tasso C. (Eds.) *Topics in Expert System Design,* North-Holland, pp. 353-376.

Preece, A. D., Shinghal, R. and Batarekh, A. (1992) Principles and practice in veryifying rule-based systems. *The Knowledge Engineering Review,* Vol 7:2, 115-141.

Preece, A. D. (1993) A New Approach to Detecting Missing Knowledge in Expert System Rule Bases. *International Journal of Man-Machine Studies.* 38, 661-688.

Price, B. A., Baecker, R. M. and Small, I.S. (1993) A Principled Taxonomy of Software Visualization**.** *Journal of Visual Languages and Computing* 4 (3), pp. 211-266.

Richer, M. and Clancey, W. (1985) Guidon-Watch: A Graphic Interface for Viewing a Knowledge-Based System. *IEEE Computer Graphics and Applications.* 5 (11), pp. 51-64.

Shadbolt, N, Motta, E., and Rouge, A. (1993) Constructing Knowledge-Based Systems. *IEEE Software 10(6)*, November, 1993.

Stasko, J. T. (1990). Tango: A Framework and System for Algorithm Animation. *IEEE Computer*, 23(9) pp. 27-39.

Stasko, J. T. and Wehrli, J. F. (1993) Three-Dimensional Computation Visualization. Proceedings of the 1993 IEEE Symposiuim on Visual Languages. Bergen, Norway, August, 1993. pp. 100-107.

Turing, A. M. (1950. Computing machinery and Intelligence. *Mind* pp. 433-460.

Yost, G. R. (1992) Configuring Elevator Systems. Workplace Integration Technologies Group, Digital Equipment Corporation, 11 Locke Drive (LM02/K11), Marlboro, MA 01752. December. Unpublished.