REFERENCES

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.

Boehm, B., W. (1987) Industrial software metrics top 10 list. IEEE software, 4 (5) September, 84-85.

Conklin, J. and Begeman, M.L. (1988) gIBIS: A Hypertext Tool for Exploratory Policy Discussion. ACM Trans. on Office Information Systems, 6(4). pp. 303-331.

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.

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.

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

Motta, E., Rajan, T., and Eisenstadt, M. (1990) Knowledge Acquisition as a process of Model Refinement. *Knowledge Acquisition,* vol. 2, pp. 21-49.

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.

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

Rajan, T., Motta, E., and Eisenstadt, M. (1988) Acquist: a tool for knowledge acquisition. in Kelly, B., and Rector (ed.) Research and Development in Expert Systems V. Cambridge University Press. pp 113-125.

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

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.

Software visualization can also 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 in section 4.1, that of observing the input-output behaviour of a subsystem.

7. SUMMARY

As I mentioned in the introduction even though this task was stopped I believe it still has managed to produce some useful results. We can view the overall result from the three related tasks Bug Location Methodology (T231), Visualization Tool (T342), and this task as a software visualization based debugging and validation framework. Basing the framework on software visualization technology is easy to defend. As Guida and Mauri [1993] say:

> “...it is straightforward to recognize that the ultimate goal of evaluation is KBS behaviour” p. 209
>
> “Measuring performance directly from the definition turns out to be a difficult task: in fact, it is hard to envisage the actual behaviour of a KBS and to compare it with the specifications.” p. 210

Support is provided via:

- the attachment of images to concepts created in the analysis or design phase,
- domain dependent editor creation allowing expert scripted visualizations to be created,
- a framework and language for creating visualization systems (see DD231 and DD342),
- traceability from implementation level visualizations to design and analysis phases.

This work will carry on in two forms. As part of the OU’s exploitation plan we will port the protocol editor, OCML editors and CLIPS visualization onto the PC. We have in fact, using internal resources, already ported the majority of the interface code. We also plan to investigate the use of software visualization to drive knowledge acquisition. An expert would create an animation by direct manipulation of a domain dependent environment. During the scripting process the KE would elicit a mapping between interface actions and tasks. A primitive tasks would map onto single interface action (or possibly a small set of interface actions). At a later stage the KE would develop parts of the PSA based on animation segments. PSA modules would be validated immediately by using them to drive the animation. This is possible because of the mapping between primitive tasks and interface actions. This would necessarily be a preliminary validation as a full validation requires an expert not directly involved in the development of the KBS - a full validation could be carried out once the KBS had been completed.

- the ability to view the code executions at coarse and fine-grained views.

The construction of visualizations was aided by three tools:

- Viz - the framework and language described in DD231 and DD342,
- the ability to link concepts within the protocol editor to icons, and
- the Viz event annotation editor - an editor which allows KE to annotate a segment of code with Viz event by direct manipulation.

A summary of the support the workbench provides in design and validation is depicted in figure 24 below.

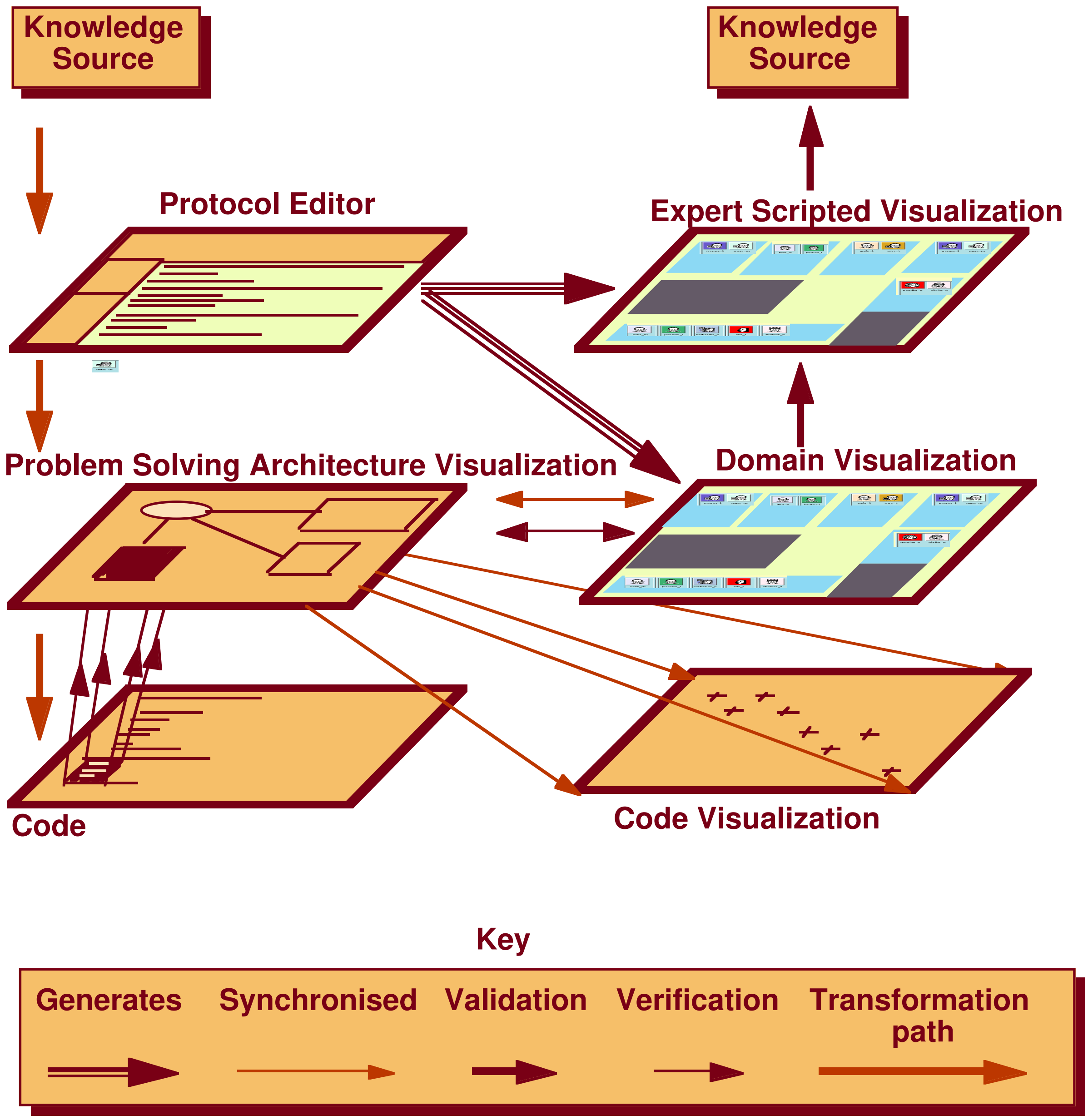

Figure 24. The SV based validation support provided by VITAL.

We can see from the example that with software visualization 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

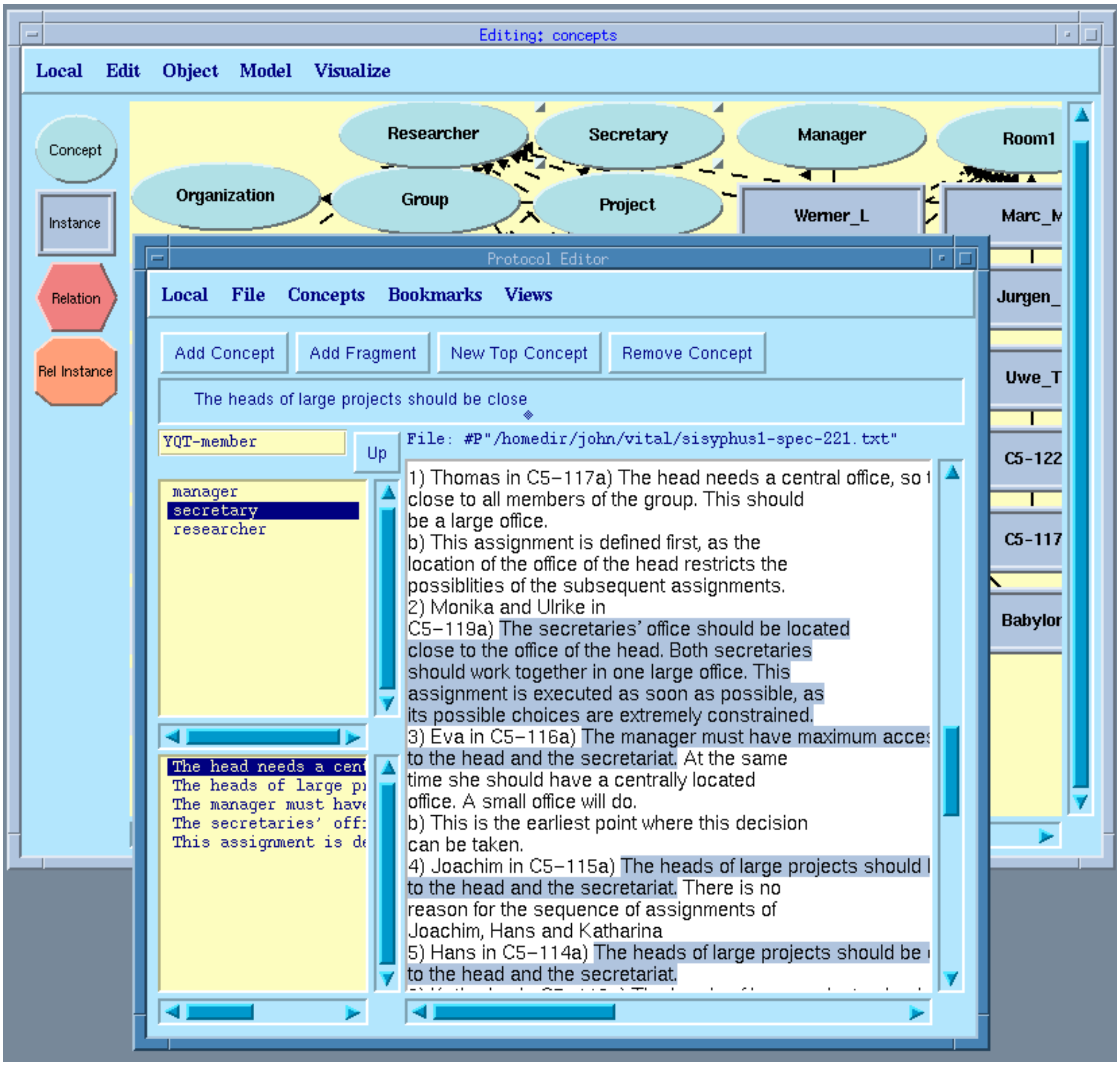

Figure 23. Tracing back from the OCML concept 'secretary' to the protocol concept 'secretary'.

6. SUMMARY OF SUPPORT PROVIDED

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 software visualization techniques providing synchronised problem solving architecture, domain and code views onto a KBS. In the example we saw how the comparison between an expert scripted and KBS driven 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 expert and KBS behaviour have a common rich representation,
- 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

```
(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 22. A fine grained view of the rule researcher-rule-1.

5.7 Traceability

As stated in section 3 one of the aims of this task was to enable tracing back from the implementation level through the design level to the analysis level. Again, because the task was stopped I was not able to complete this software module. As shown in the previous sections the implementation and design levels are tightly linked. The links between the design and analysis level are weak. It is possible though to move back from an OCML concept to the corresponding concept within the protocol editor. This is shown in figure 23 below.

5.6. Code Level Visualizations

Figure 21. A code level view of allocate-researchers task.

Figure 21 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 22 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 21) 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.

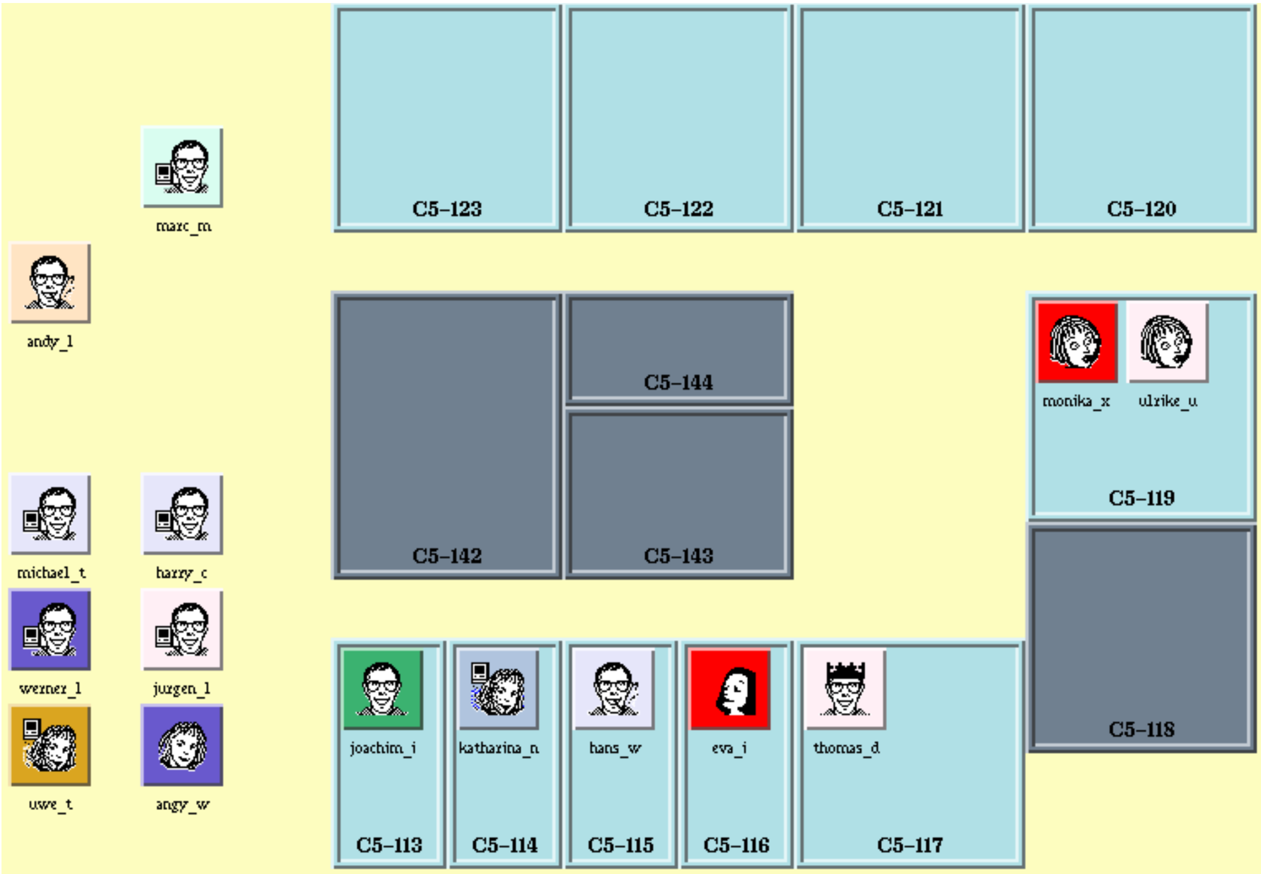

Figure 20. The domain level visualization just before allocate-researchers is executed.

Figure 19 above shows the PSA diagram when the replay halts. Figure 20 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 18. 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 18, 19, and 20 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.

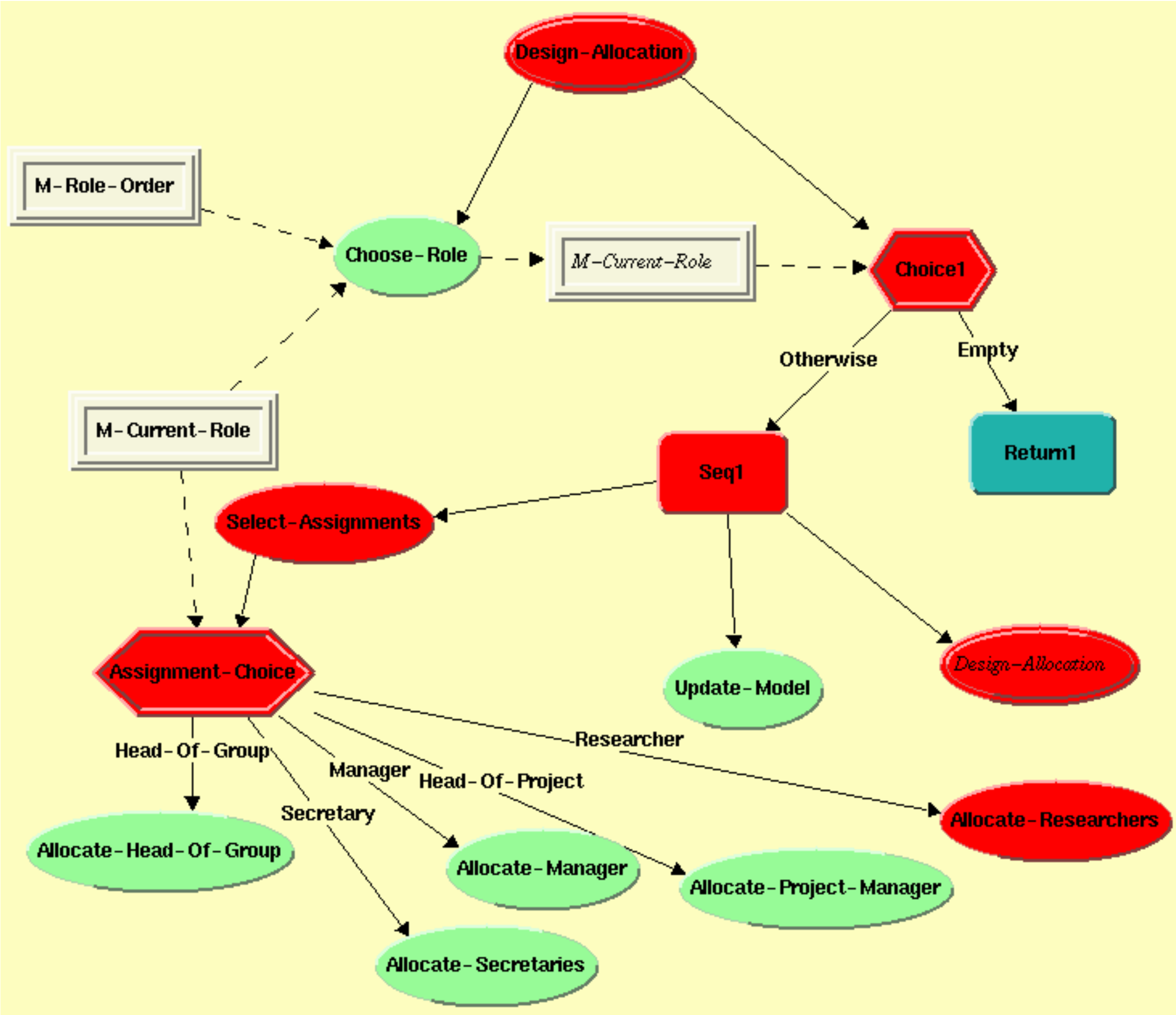

Figure 19. The PSA diagram after replaying to the paused point.

5.5.2. The Sisyphus I Domain Visualization

Figure 18 below shows the domain oriented visualization at the end of replay.

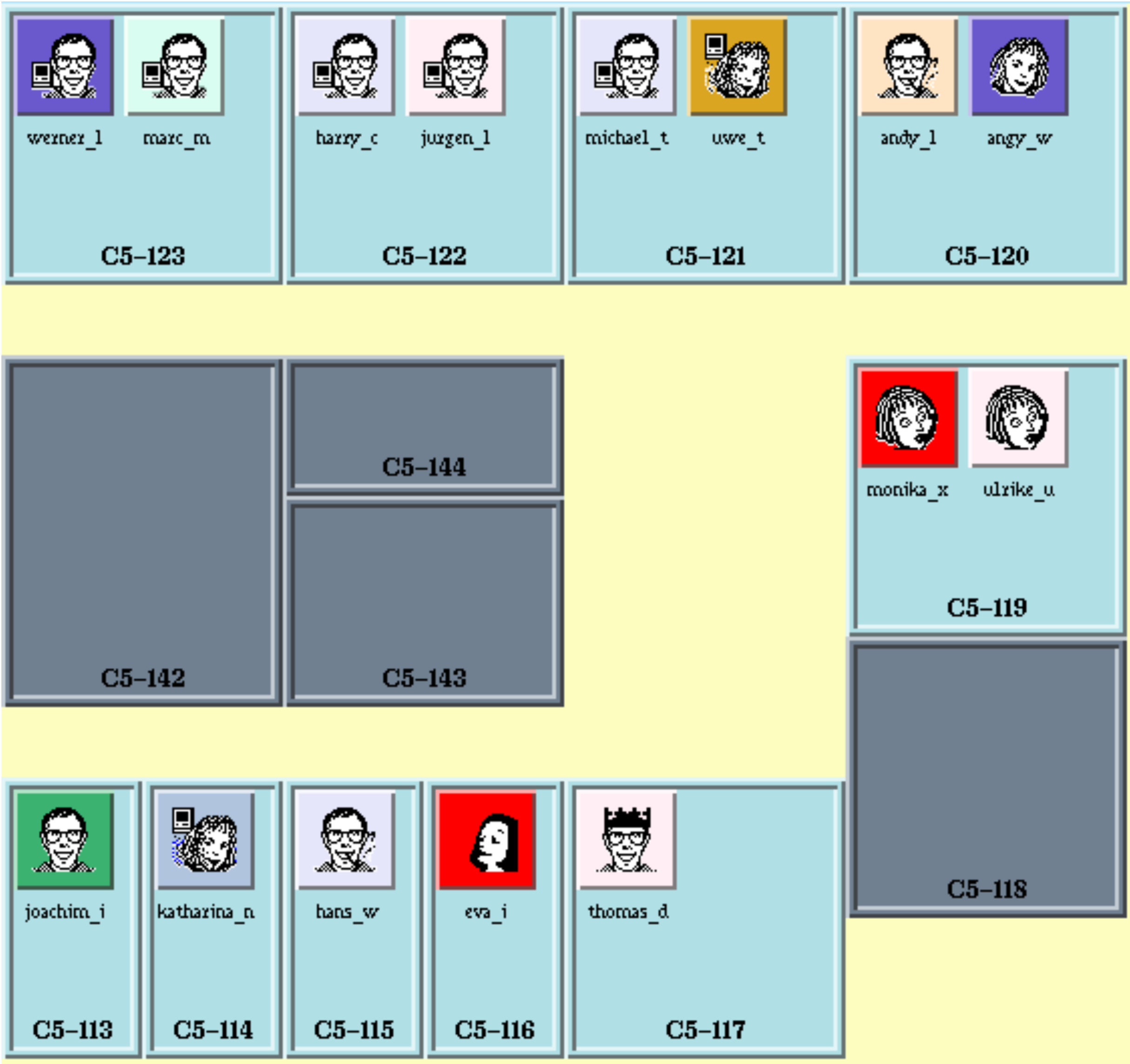

Figure 18. The Sisyphus I domain oriented visualization at the end of execution.

As the visualization runs the icons slide into the rooms. We can see by comparing figures 16 and 18 that the top row is different - the fourth constraint has been violated by the KBS. The rooms C5-120 and C5-121 contain a smoker and non-smoker. Notice that a mismatch between expert behaviour and the KBS behaviour is extremely easy to see 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.

- the ability to link concepts within the protocol editor to icons (this can also be carried out using the OCML domain editor) described in section 5.2, and
- the Viz event annotation editor - an editor which allows KE to annotate a segment of code with Viz event by direct manipulation.

For explanation purposes here is a brief summary of the Viz framework:

- *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. Views may be borrowed from the scenarios created in an expert scripted visualization.
- *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*. The mappings can borrow from the icons created in the expert scripted visualization.
- *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.

Once a visualization has been defined using the metaphor of players, states, events, views and mappings, it can be attached to the implementation layer using Viz event annotation 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.

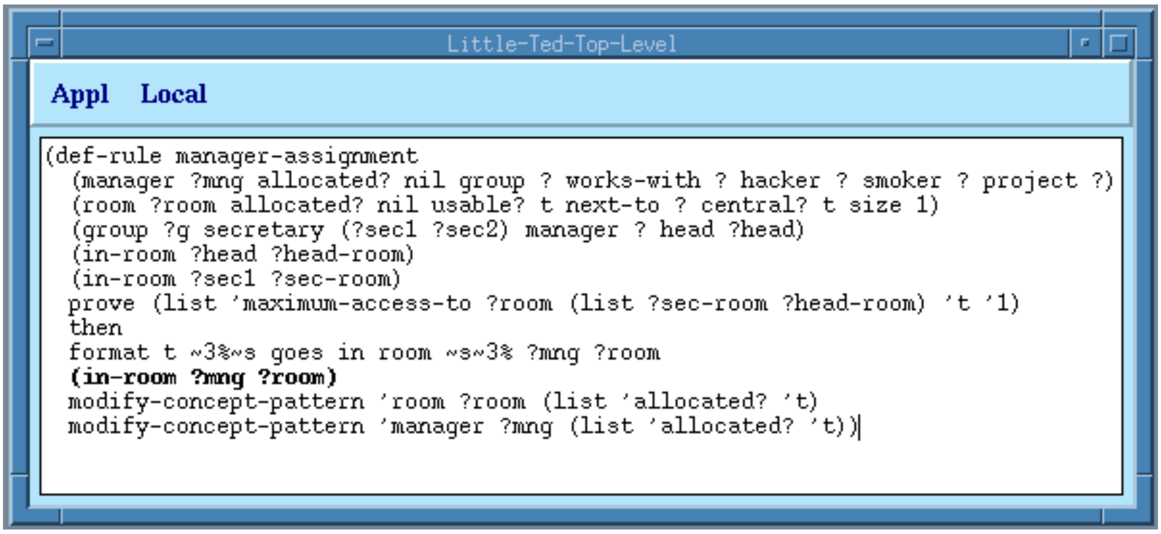

Figure 17. An annotated Sisyphus I rule for the domain visualization.

Again because the task was stopped the annotation editor was not completed.

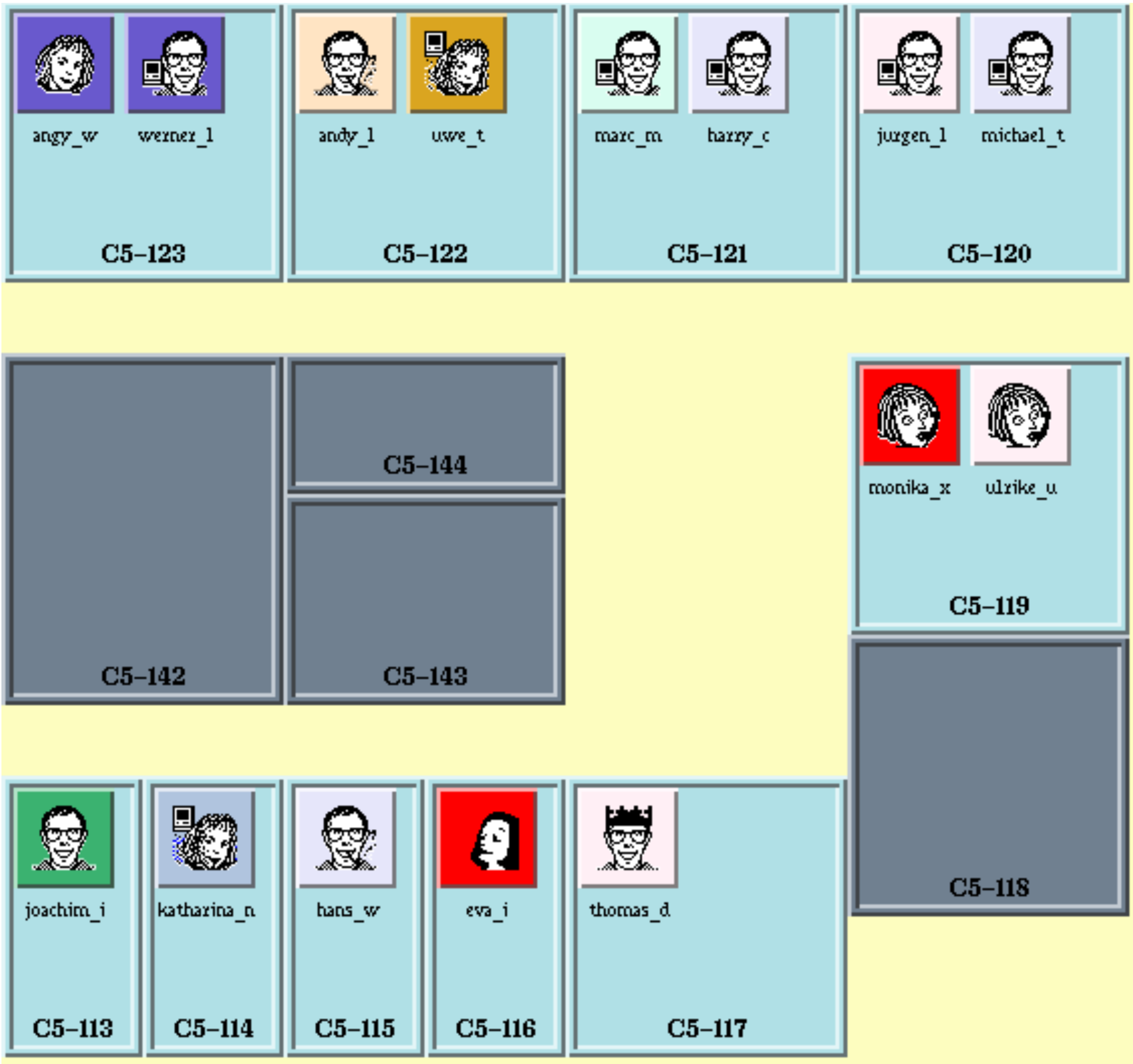

Figure 16. The final state of an expert scripted domain visualization.

Within the visualization shown in figure 16 each icon represents a member of the group. The colour of each icon indicates the current project of the member. This can be compared with the domain oriented visualization described in the next section.

It should be emphasised here that the core conceptual contribution of this task is the expert scripted visualization.

5.5. PSA and Domain Oriented Visualizations

As stated in section 2.1 (the AC pilot visualization) the PSA and domain oriented 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 (based on the expert scripted visualization) and a predefined TRI style (described in section 2.2) visualization were used.

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. There are three tools which can aid in the construction of visualizations:

- Viz - the framework and language described in DD231 and DD342,

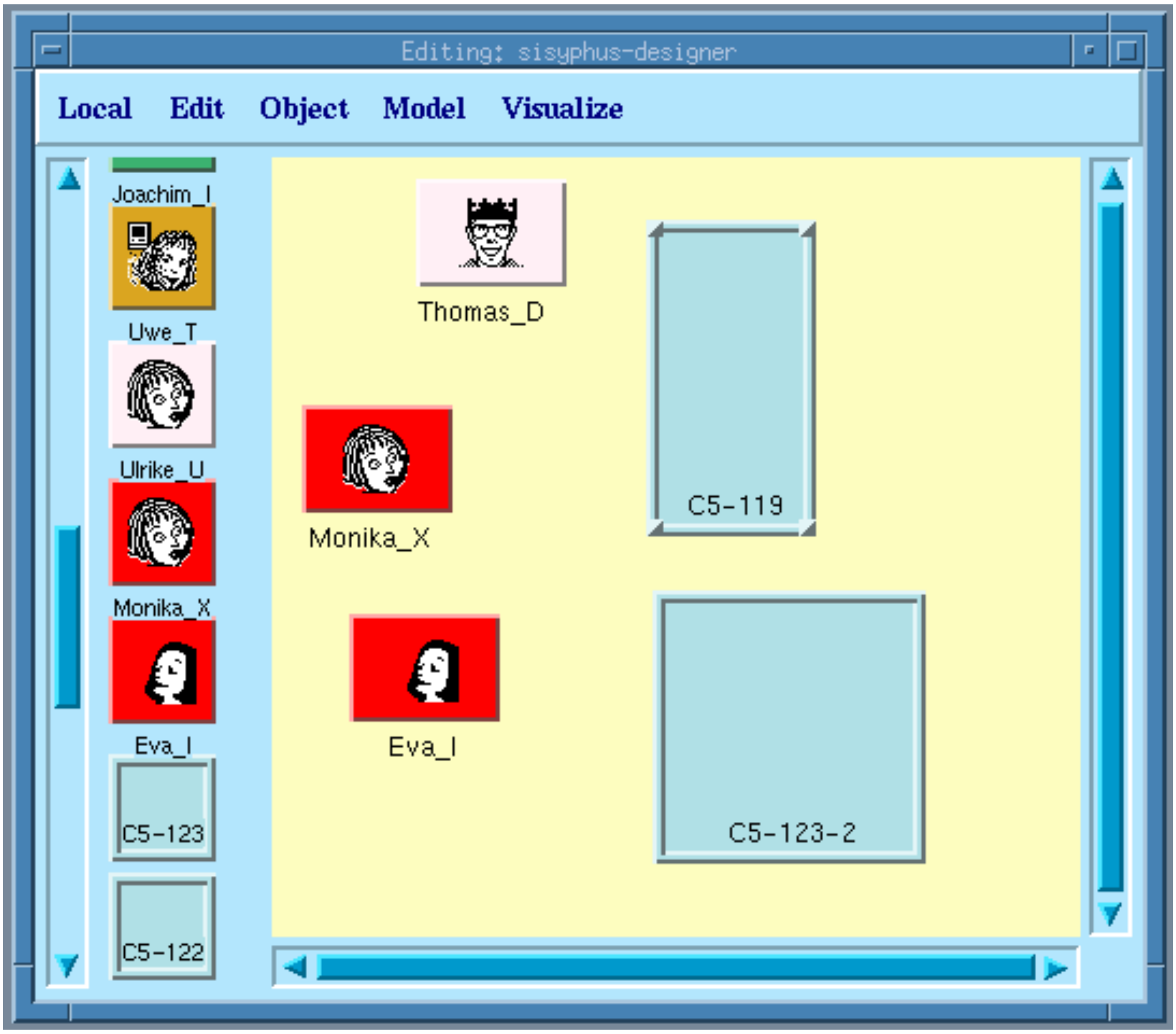

Figure 15. A snapshot of a Sisyphus I script editor. The palette on the left is a well containing all the YQT researchers and the rooms.

Using the above editor a scenario 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. Figure 16 below shows the final state of an expert scripted domain visualization:

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.

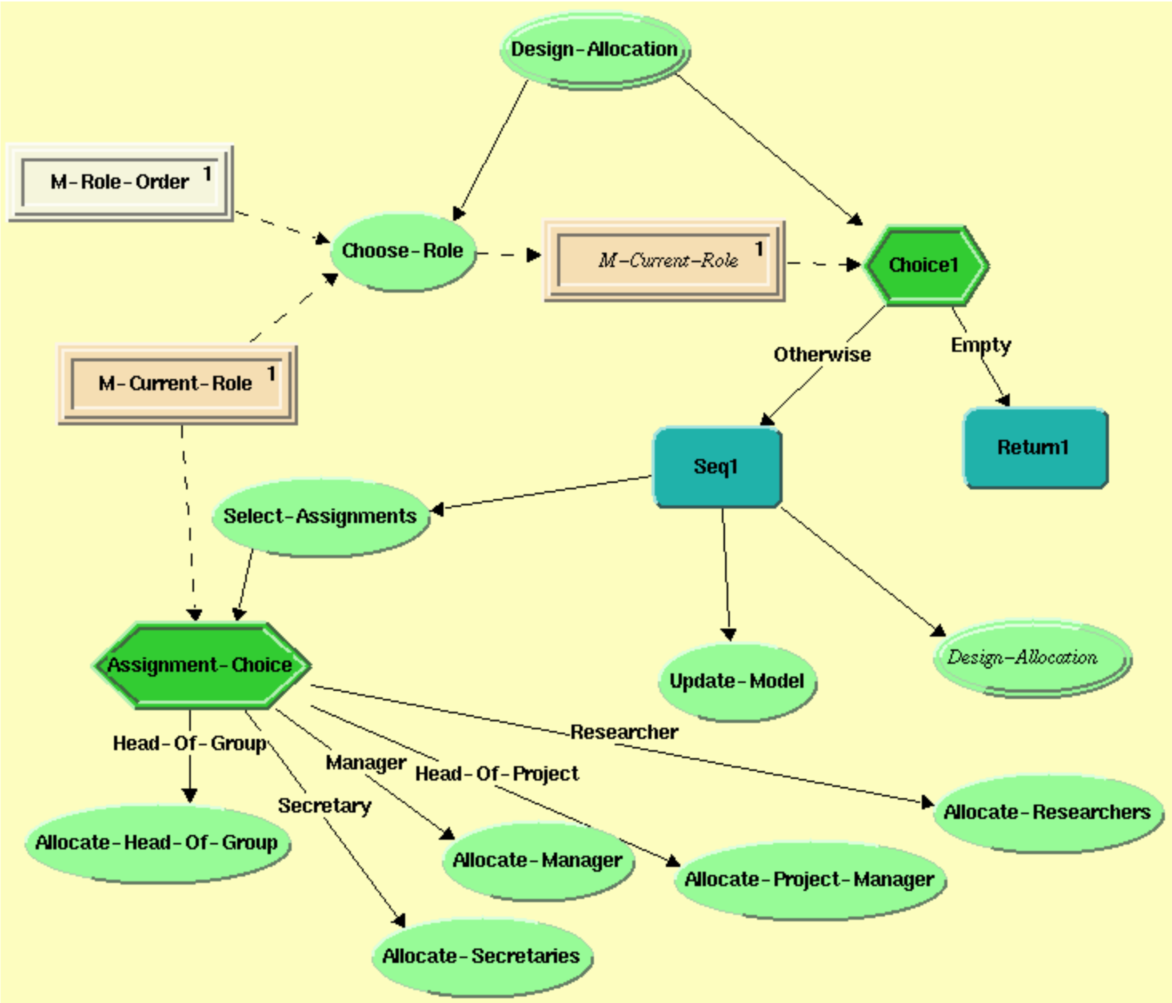

Figure 14. The Sisyphus I problem solving architecture.

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.

5.4. The Expert Scripted Visualization

As I stated in section 4.1 there are two problems with Turing test based validation. Firstly, it is hard to make the output of a KBS resemble human dialogue. Secondly, a typescript is not interactive. The solution to the problem has been to use software visualization as a common rich description language for the expert and KBS. In conjunction with a KE the domain expert can create an Expert Scripted Visualization (ESV). Once a set of mappings have been created (within the protocol editor) a specialised Sisyphus I script editor can be created. A snapshot of one in use is shown in figure 15 below.

The images can be in either a GIF or PPM format. With existing converters this means that any image which can be displayed within X11, MicroSoft windows or on a macintosh can be used.

Once the mapping has been completed we can generate OCML concepts, instances, relations and relation instances. The mapping between the concepts created within the protocol editor and OCML concepts is defined by the KE. Unfortunately, because this task was halted the mapping from the protocol editor to OCML has not been completed.

5.3. The Sisyphus I Design in VITAL

The Sisyphus I design in VITAL consists of three modules. The first module contains the relations such as ‘being within a room’. The second comprises the concepts and instances such as the researchers and room. Figure 13 below shows the graphical representation of the second module.

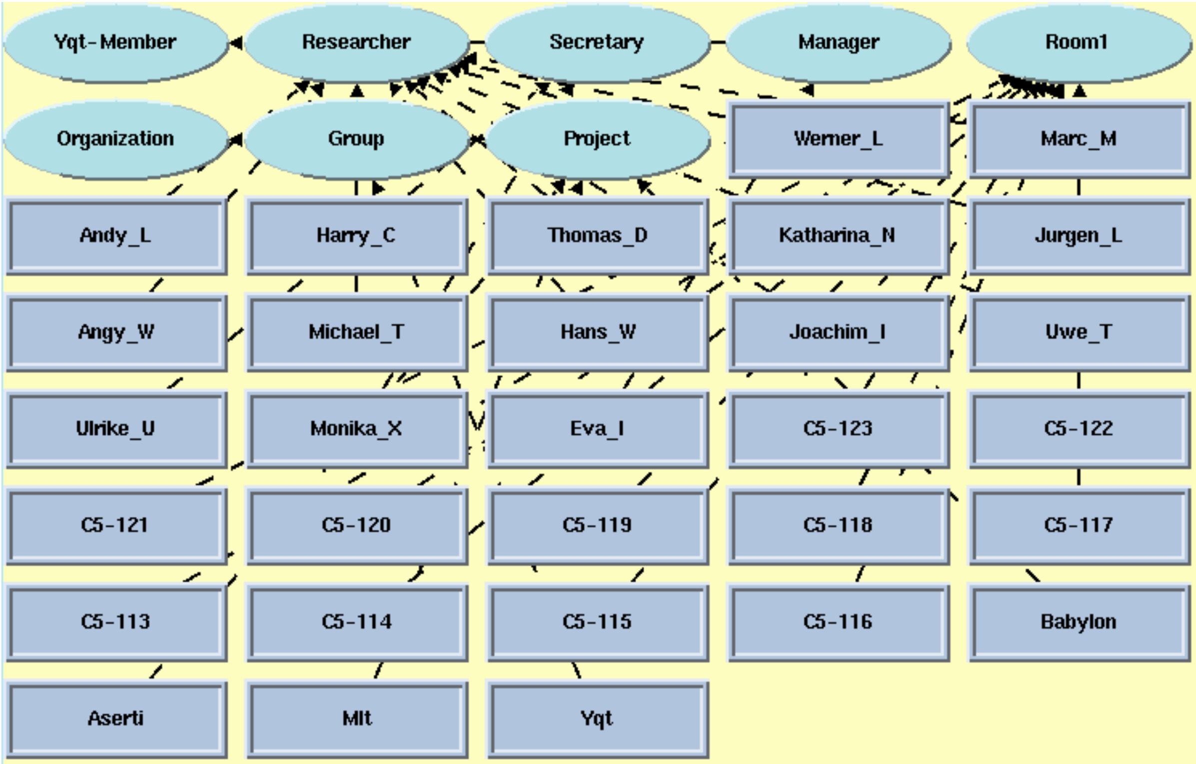

Figure 13. The Sisyphus I concepts and instances represented in OCML.

The concepts are shown as ovals (blue on a colour screen) and the instances are shown as framed rectangles. All the instances are either people (e.g. Monika_X) or rooms (e.g. C5-120).

The third and final module contains the problem solving architecture. Figure 14 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 below. The hexagonal node is a choice control construct. In a choice construct control flow

special purpose menu the KE creates a mapping between concepts and graphical icons. The complete Sisyphus I mapping is shown below:

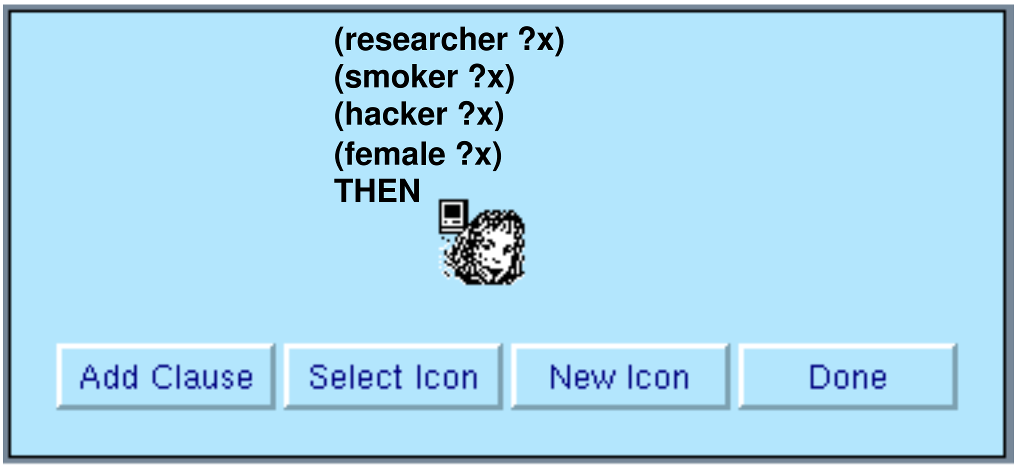

Figure 12. A snapshot of the menu used to create mappings between concepts and graphical icons.

The menu above indicates that female researchers who smoke and program are represented by the  icon.

The full set of mappings is shown below:

-  or  - researcher,
-  or  - smoker,
-  or  - hacker,
-  or  - hacker and smoker,
-  - manager,
-  - secretary,
-  - head of group.

- room.

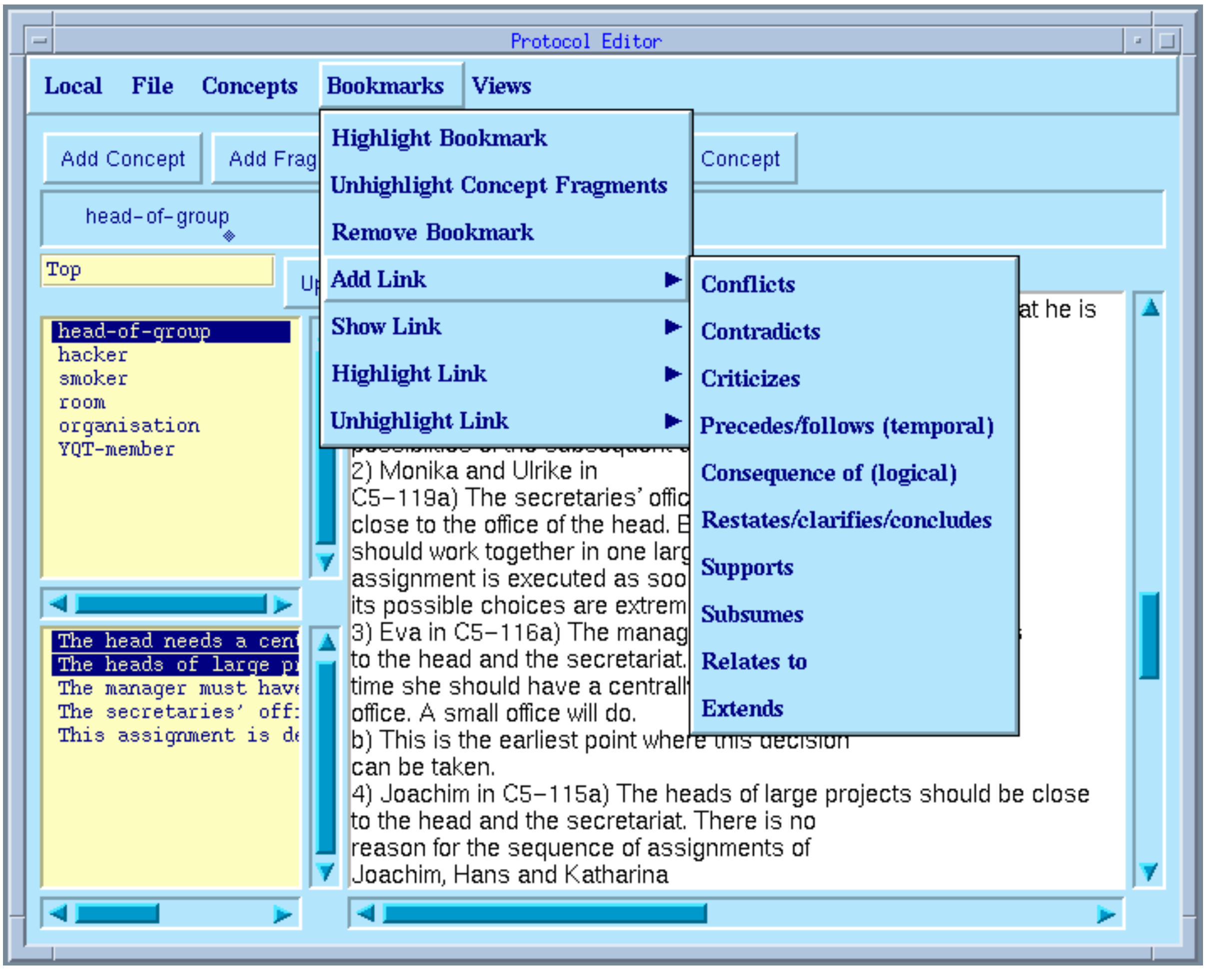

Figure 11. The types of bookmarks allowed.

These types are from IBIS literature such as [Conklin & Begeman, 1988]. The meaning is as follows:

- conflicts - fragment A conflicts with fragment B,
- contradicts - fragment A contradicts fragment B,
- criticizes - fragment A criticizes fragment B,
- precedes/follows (temporal) - fragment A occurred before fragment B,
- consequence of (logical) - fragment A is a logical consequence of fragment B,
- restates/clarifies/concludes - fragment A restates the concepts in fragment B,
- supports - fragment A is a supporting piece of argument for fragment B,
- subsumes - fragment A contains a superset of the concepts in fragment B,
- relates to - fragment A and fragment B are related somehow,
- extends - fragment A extends the concepts in fragment B.

As well as fragments of text, images can be associated with concepts. The small window in the bottom left of figure 7 shows an image associated with the concept head-of-group. Images associated with a concept can be used to define domain level visualizations. Using a

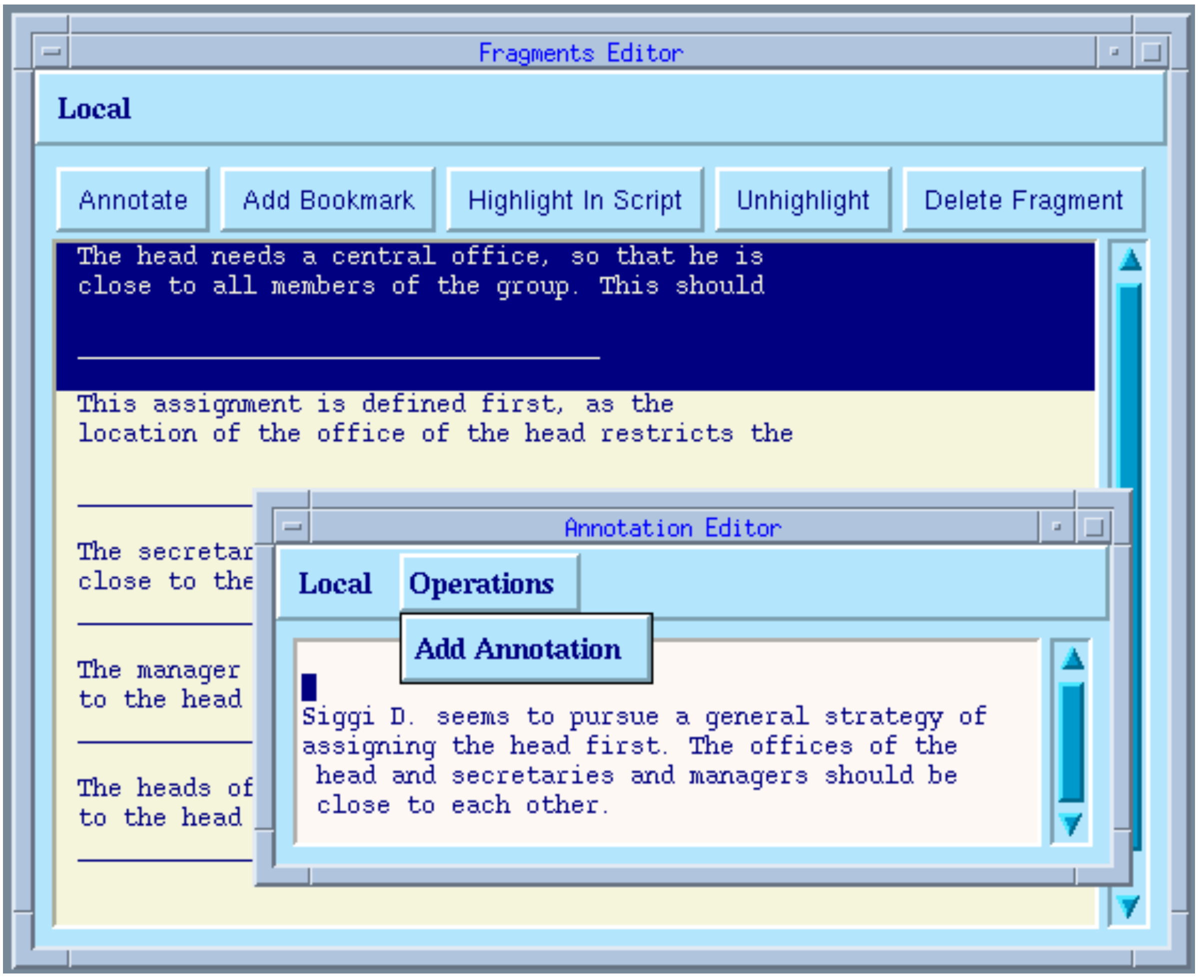

Figure 10. The annotation editor.

As figure 11 below shows it is possible to carry out link operations between bookmarks.

Using the “Concepts” menu it is possible to highlight the fragments associated with a concept or to view them in a fragments editor. The fragments which overlap with a highlighted region of text can also be viewed in the fragments editor. A screen snapshot of the fragments editor is shown below.

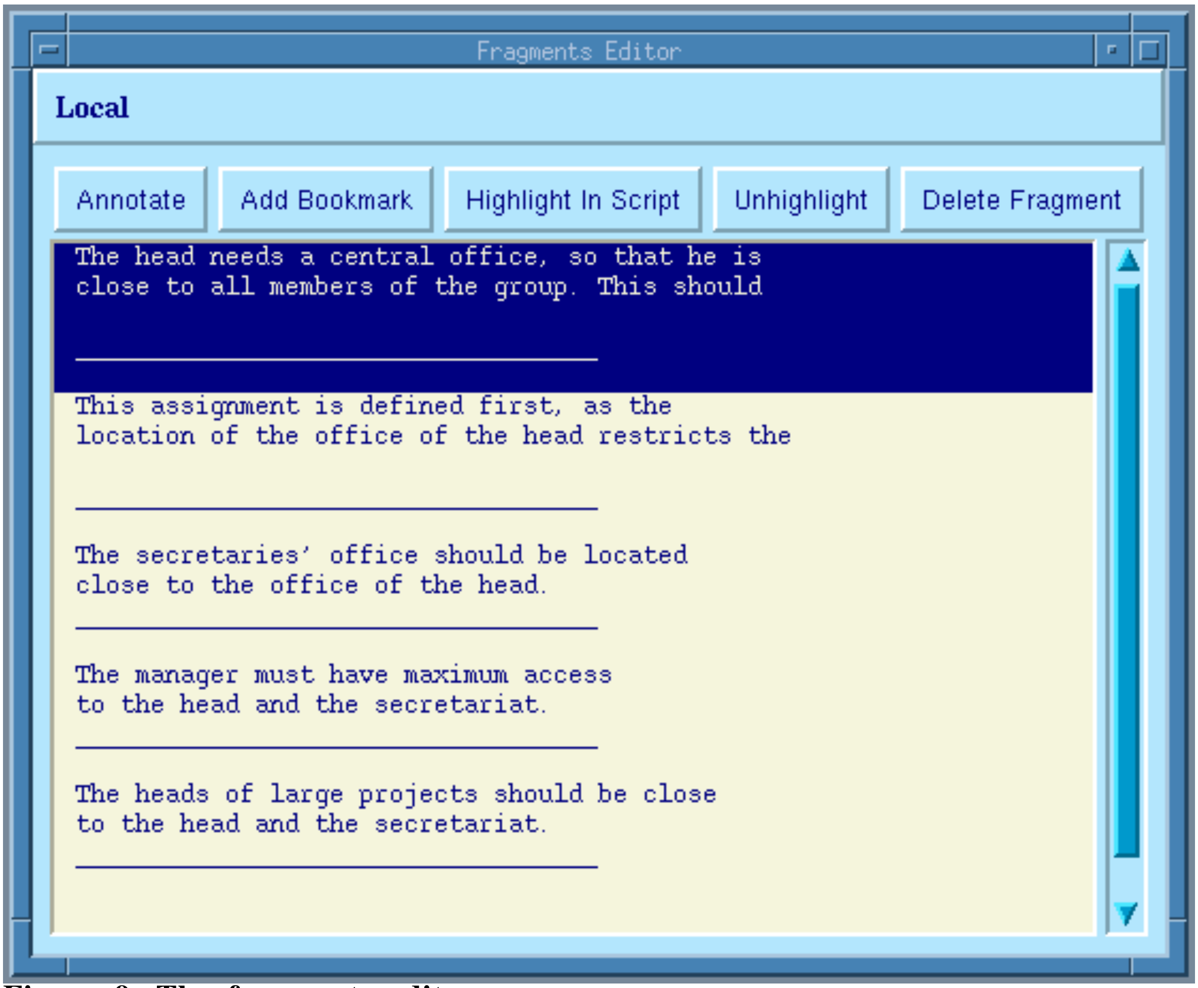

Figure 9. The fragments editor.

The figure above shows three fragments from the concept “head-of-group”. Using the fragments editor a fragment can be:

i) annotated,

ii) added to the list of bookmarks (shown in the bottom left window of figure 7),

iii) or highlighted within the protocol editor.

When a fragment is annotated an annotation editor pops up. This is shown in the figure below:

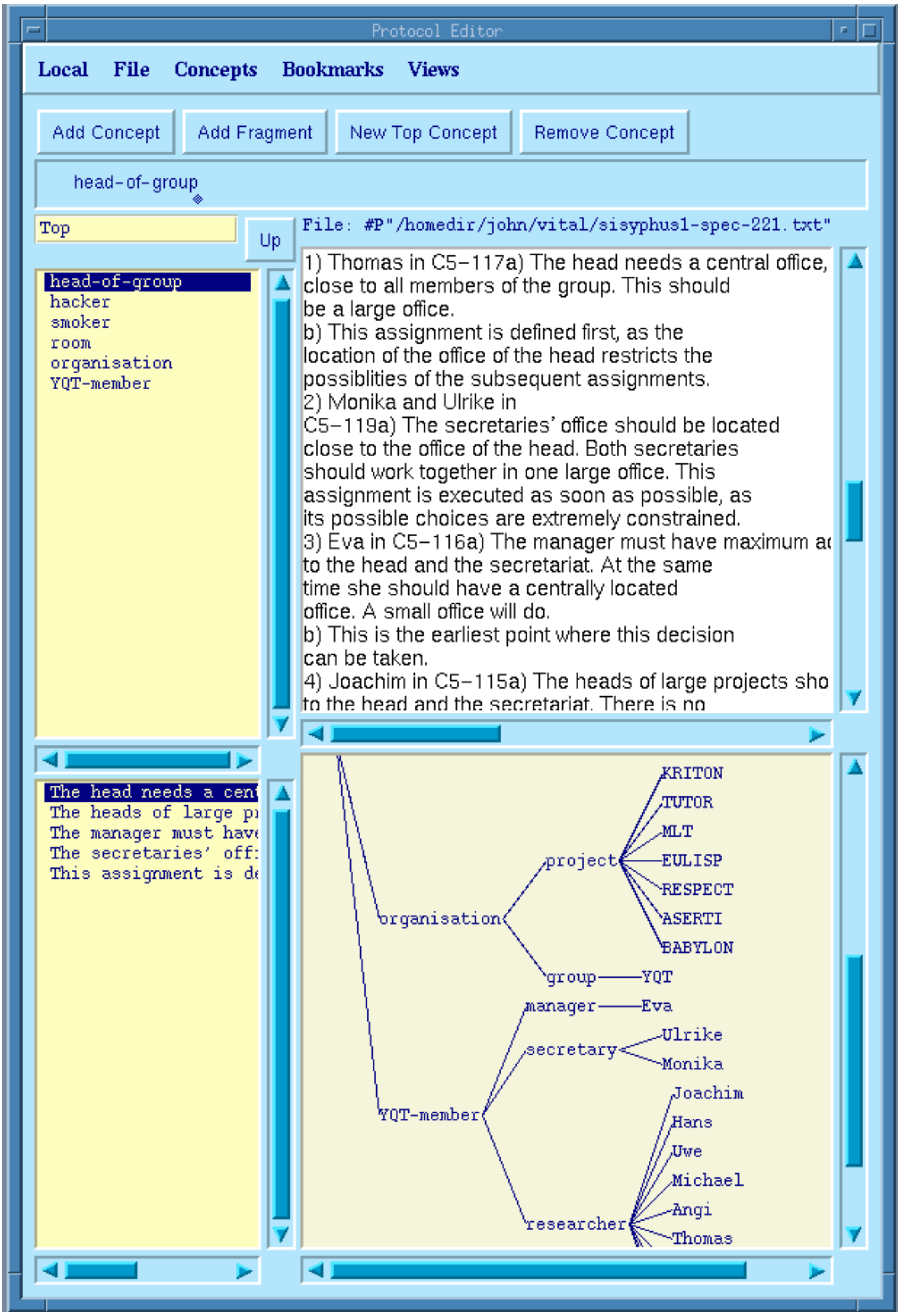

Figure 8. The protocol editor showing a graph of the concept hierarchy.

We can move around the concept hierarchy by clicking on a concept within the graph.

5.2. Sisyphus I Protocol Analysis

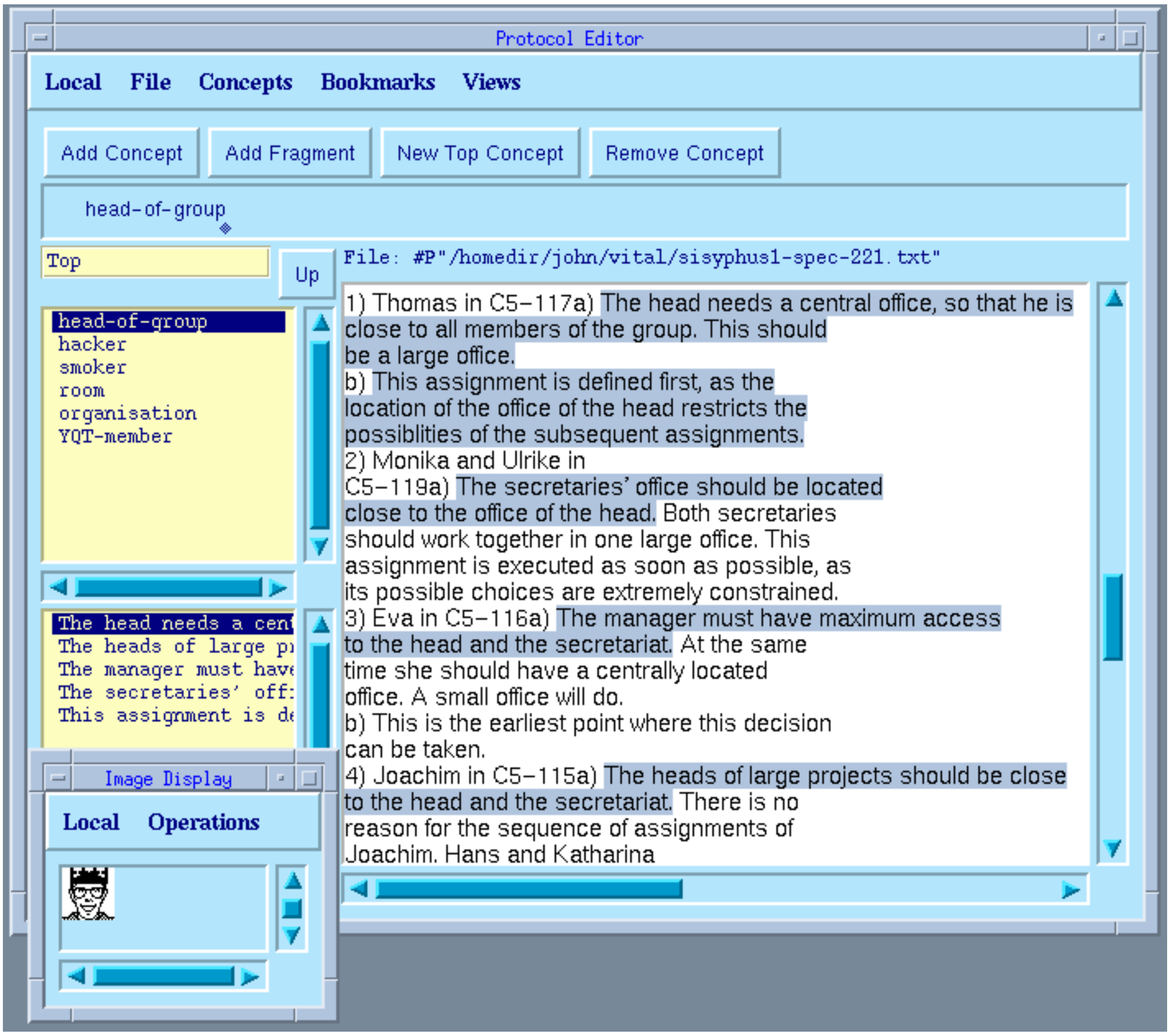

Figure 7. The protocol editor

Figure 7 above shows the protocol for Sisyphus I within our protocol editor (created within this task). The protocol editor has a functionality similar to Acquist [Rajan et al., 1988; Motta et al., 1990]. The top scroll window on the left of the figure contains concepts (head-of-group, hacker, smoker etc.). Fragments of text can be selected in the editor window on the right and either be associated with a concept or added to a list of bookmarks (or both). The bookmarks are shown on the bottom scroll window on the left ('The head needs a cen..', 'The heads of a large p..' etc.). The text window on the right of figure 7 shows part of the protocol where the fragments associated with head-of-group have been highlighted.

A segment of text can be highlighted by dragging the mouse - clicking on a word whilst holding the meta key down highlights the word. The highlighted text appears within the small edit window (currently containing "head-of-group"). This allows the knowledge engineer to edit a concept or fragment before it is stored. The highlighted text can be converted into a fragment or concept by clicking on the appropriate button. New concepts are stored as a child of the current concept - shown in the small window above the list of concepts. In the figure we have just added the concept "head-of-group". We can move up the concept hierarchy by clicking on the "Up" button. It is possible to obtain a graphical view of the concept hierarchy as shown in figure 8 below.

Software visualization can aid face validation by providing a ‘user-friendly’ medium for an expert. The method used to facilitate Turing test based validation has become the cornerstone of the bug location framework. The main idea is to view Turing test based validation as a human-computer interaction problem. The ideal is to provide a medium that is ‘rich’ and ‘natural’ for non-computer-literate-experts to express themselves whilst remaining within reach for KBS. This is discussed in detail in section 6. I shall now illustrate the framework using the Sisyphus I example.

5. THE SISYPHUS I EXAMPLE

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 use the Sisyphus I example to explain the bug location framework and tools. The tools support the framework in two ways. Firstly, they enable *traceability* from the implementation layer to the analysis layer. Secondly, they support the *creation* of visualizations which in turn enhances Turing test style validation. The framework is described in more detail in section 6.1.

4. INFLUENCES FROM VALIDATION AND VERIFICATION

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

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. There are cases when formal techniques can not be used. For example, when an ad-hoc mixture of knowledge representations are used. In these cases a comparison needs to be made between human performance and system performance. The problem of comparison can be viewed as a human-computer interaction one and are therefore targettable by software visualization techniques.

4.1. Suitable Validation and Verification Techniques

Certain validation techniques lend themselves to being aided by software visualization. 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. There are two problems for validation based on human-computer comparisons. First, the performance outputs from humans and computers are very different - like is not compared with like. Second, there is the problem of human evaluators' bias against computer systems, as occurred in evaluations of the medical diagnosis system MYCIN [Gaschnig et al., 1983]. The solution to these problems is to use recordings of performance in which the source is unidentifiable - a 'recorded Turing Test'. This is described below.

- 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. 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.

- 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.

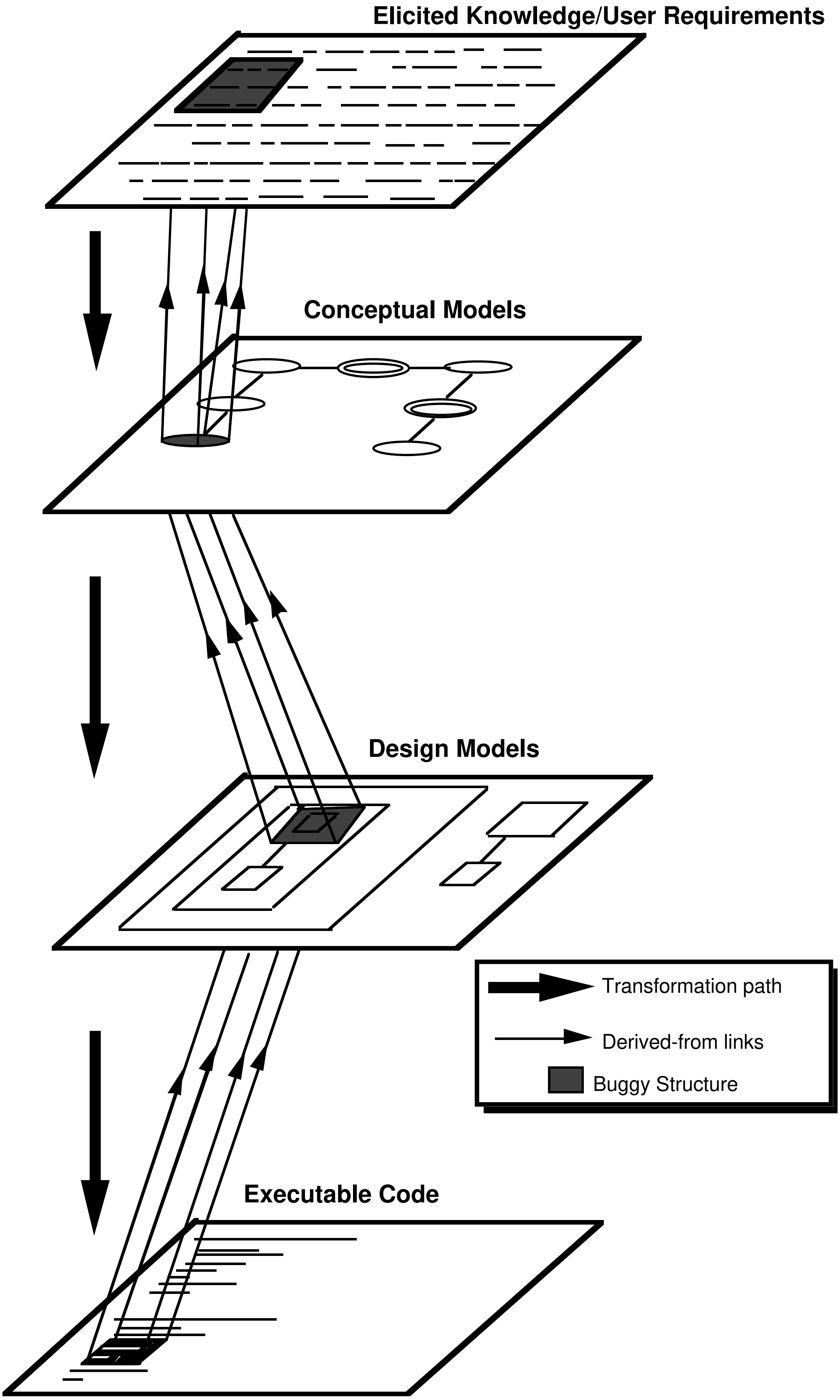

Figure 6. The use of 'derived from' links to aid in inter-process product debugging.

I shall describe how we have facilitated this inter-process product debugging in section 5.

The second study analysed the debugging anecdotes collected from a world-wide email trawl which revealed three primary dimensions of interest: *why the bugs were difficult* to find, *how* the bugs were found, and *root causes* of bugs. Half of the difficulties arose from just two sources: (i) large temporal or spatial chasms between the root cause and the symptom, and (ii) bugs that rendered debugging tools inapplicable. Techniques for bug-finding were dominated by reports of data-gathering (e.g. print statements) and hand-simulation, which together accounted for almost 80% of the reported techniques. The analysis pinpointed a winning niche for VITAL bug location tools: data-gathering or traversal methods to resolve large cause/effect chasms which made some of the bugs so hard to locate. The need for data-gathering and traversal tools lead to the construction of Viz, a framework and software tool to describe and implement software visualization systems, which was described in DD231 and DD342.

The cause-effect chasm (the gap between where a bug's symptom is viewed and its cause) is exasperated by the fact that the VITAL knowledge engineering methodology is based around the successive refinement of the four process products: user requirements, conceptual model (CM), design models (DM), and executable code (EC). Due to this sequential process, any bug that is introduced in a particular model, will be inherited by successive ones where it may manifest itself. For example, an incorrect OPS5 rule may be the result of an erroneous piece of knowledge elicited from the expert, and then correctly transformed into a conceptual and design model. In order to locate the true source of the bug a link from the buggy rule to appropriate parts of the design and conceptual models and to the elicited piece of knowledge must be present. To enable this inter-process product debugging the aim in this task has been to keep ‘derived from’ links between the process products as shown in the figure below.

```
Rule: drop-object
Definition:
?goal <- (goal-is-to (action hold) (arguments blank))
?monkey <- (monkey (location ?place) (on-top-of ?on) (holding ?name&~blank))
?thing <- (thing (name ?name))
=>
(printout t "Monkey drops the " ?name "." crlf)
(modify ?monkey (holding blank))
(modify ?thing (location ?place) (on-top-of ?on))
(retract ?goal)
matches for 1
  nil
matches for 2
  (monkey (location t2-2) (on-top-of floor) (holding red-key))
matches for 3
  (thing (name green-couch) (location t5-7) (on-top-of floor) (weight heavy))
  (thing (name red-couch) (location t2-2) (on-top-of floor) (weight heavy))
  (thing (name big-pillow) (location t2-2) (on-top-of red-couch) (weight light))
  (thing (name blue-chest) (location t7-7) (on-top-of ceiling) (weight light))
  (thing (name blue-couch) (location t8-8) (on-top-of floor) (weight heavy))
  (thing (name green-chest) (location t8-8) (on-top-of ceiling) (weight light))
  (thing (name red-chest) (location t2-2) (on-top-of floor) (weight light))
  (thing (name red-key) (location held) (on-top-of held) (weight light))
  (thing (name ladder) (location t2-2) (on-top-of red-chest) (weight light))
partial matches for (1 - 2)
  none
partial matches for (1 - 3)
  none
activations
  none

in cycle: 29
```

Figure 5. A screen snapshot of Drop-Object showing working memory matches.

We can see from figure 5 above that the reason the rule failed to match working memory at cycle 29 was that there were no working memory elements which matched (goal-is-to (action hold) (arguments blank)).

3. INFLUENCES FROM THE BUG LOCATION METHODOLOGY

As stated in the introduction one of the major influences in this task was the output of task T231 Bug Location Methodology. The final report for task T231 reported on two studies which were aimed to provide concrete input from ‘a state of art programming environment’ and from ‘the real world’. The first study took the form of an in-depth self-report. This study revealed eight fundamental problems in the debugging facilities provided by the programming environment: indirect access to troublesome source code; disruptive intermediate actions required; poor interpreter access during breaks; poor monitoring of built-in functions; no coarse-grained view of execution; no data flow analysis; no control flow analysis; deceptive view of inner states.

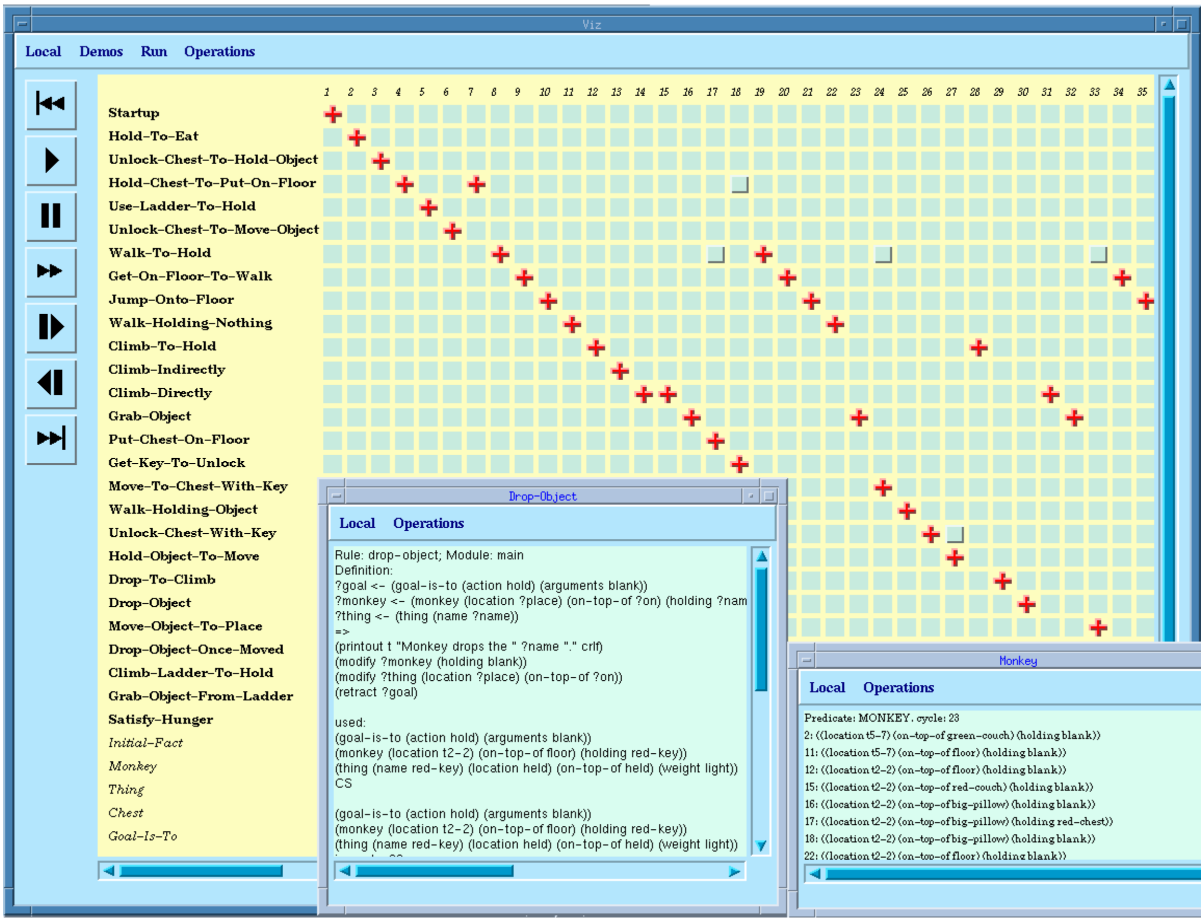

Figure 4. A screen snapshot of the CLIPS visualization (running the monkeys and bananas example). The first column shows the CLIPS rules (in bold) and predicates (in italics) used within the execution. The  indicates that rule fired, a  indicates that a rule matched against working memory creating one or more instantiations but did not fire, and a  indicates that the rule failed to match working memory (note the  and symbols are more easily distinguishable on a colour screen).

The top window in figure 4 shows the TRI style visualization (described in DD342) of CLIPS. The first column shows the CLIPS rules (in bold) and predicates (in italics) used within the execution. The smaller window on the left shows a fine-grained view for the rule Drop-Object at cycle 30, when it fired. It was created by selecting the  symbol in the 'Drop-Object' row at column 30 with the meta-key held down. The fine-grained view shows the definition of the rule, the working memory elements that it used, and all the instantiations the rule had in the conflict resolution set at cycle 30. The smallest window on the right shows a slice of working memory for the predicate "Monkey" at cycle 23. It was created by selecting the symbol (again with the meta key held down) in the 'Monkey' row at column 23. Each line within this window shows a working memory element with its creation cycle.

A fine-grained view for the rule Drop-Object at cycle 29, when it failed to match working memory is shown below.

for a function call have been evaluated the instantiated call is shown. User defined function calls (such as (select-criteria (costs-number-of-flexible-length))) are shown in a bold font and in-built function calls are shown in a plain font. We can see from the trace that select-criteria returns the first element of the list *selection-criteria* if the list has a value.

2.2. The CLIPS Visualization

As part of the OU's exploitation plan we are producing two new courses which we believe will generate over £125K per annum. One of the courses, Intelligent Systems: Knowledge Representation and Implementation, will use our PC implementation of the workbench which generates CLIPS code. As part of our exploitation work (before this work was stopped) I constructed an initial CLIPS visualization within the Unix version of VITAL.

CLIPS is a popular public domain OPS5 based knowledge engineering toolkit written in C. It was developed by the Software Technology Branch of the Information System Directorate at NASA/Johnson Space Center. It is available via anonymous ftp from ftp.ensmp.fr (192.54.148.100). The CLIPS sources have been augmented so all interesting (to Viz) CLIPS events are captured. A Knowledge Engineer (KE) can start/stop event capture by typing (viz-on)/(viz-off). It should be noted that the events capture is built on top of existing CLIPS debugging/tracing code and therefore has a negligible effect on the speed of execution.

As figure 4 below shows, the CLIPS visualization below is similar to the other OPS5 visualizations (called TRI (see DD342)) constructed with Viz.

usable but they were not the cheapest line available. The name of the twisted pair considered is displayed beside the trunk. The dark grey indicates that a line from the trunk was allocated to the customer. The name of the twisted pair chosen is displayed beside the trunk. We can see from figure 2 that the top four trunks were not usable. The line P9 from TB2 to the exchange was considered but this route was more expensive than the one containing the line P6 from TB1 to IP1 and the line P38 from IP1 to the exchange.

As the visualization is replayed the individual trunks and exchange boards change colour. The corresponding executing task is also highlighted within the PSA. Figure 3 below shows a code level visualization for the task select-criteria. Again, the software allowing the knowledge engineer to select a task and view its execution at the code level was created within this task.

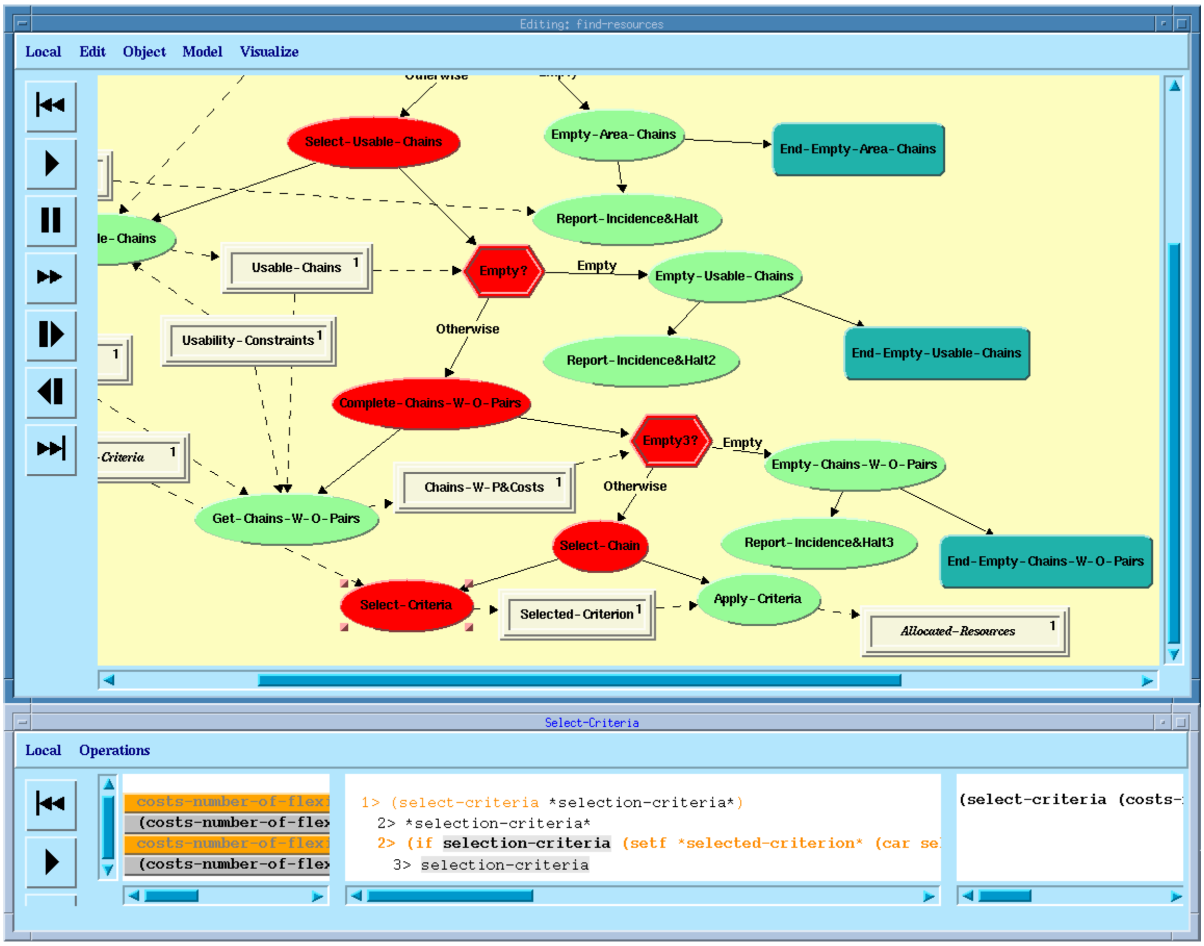

Figure 3. A PSA visualization with a code level visualization for the select criteria task.

The AC pilot was implemented in Common Lisp. The window at the bottom of figure 3 shows the Lisp tracer implemented in task T342 and described in DD342. The window was created by selecting the 'select-criteria' task (bottom left of the PSA) and choosing "Show Code Level Visualization" from the visualize menu. The Lisp tracer uses three panes. The forms to be evaluated are displayed in the second pane. The “yet to be evaluated” parts of the form are displayed in orange (light grey in this document). The parts of a form which have been evaluated are displayed in black. The current focus is highlighted in a light grey. Special forms, such as 'if', are displayed in a bold font. The first pane shows the value a form will or has returned. If the form has yet to return a value the value is shown in orange (light grey in this document). When a form has been evaluated and a value returned the value is displayed in black. The third pane shows instantiated calls. When all the arguments

The domain level visualization is contained within the window on the top right of the screen. Notice that the task diagram (which we also term Problem Solving Architecture diagram (PSA)) in the top left has a replay panel instead of a well of OCML[1] icons. The replay panel can be used (as shown in figure 1) to synchronously replay the task diagram and the domain level visualization. Previously only live visualizations of task diagrams were provided. The software to replay task diagram visualizations and to have these tied to domain level and code level visualizations (see figure 3) was created in this task. The bottom window, in figure 1, shows another portion of the task structure being highlighted as the design is replayed.

Figure 2 below shows a screen snapshot of application showing the domain level visualization.

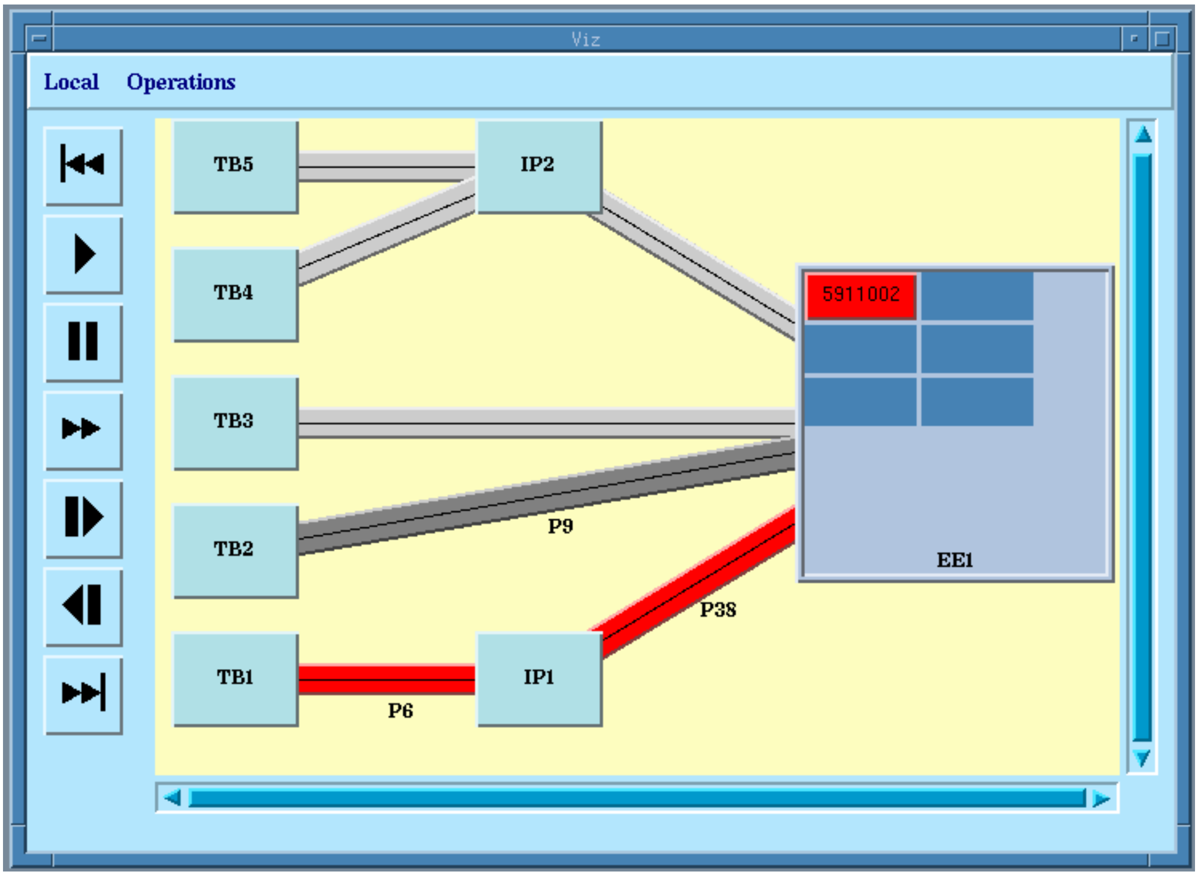

Figure 2. The AC pilot domain level visualization.

Within the domain level visualization shown above the “TB” boxes represent terminal boxes to which a new customer line is attached. “IP” boxes are used to connect 'trunk lines'. The large box on the right represents an exchange. The six rectangles within the exchange represent boards. Each board contains a sequence of numbers one of which will be assigned to the new client. We can see from figure 2 that the customer has been assigned the number 5911002 from the first board.

The thick lines between the boxes are trunks - collections of twisted pairs (of wires). The colours of the trunks indicate how one or more of the twisted pairs within the trunk were operated on by the application. The light grey indicates that none of the twisted pairs within the trunk were usable. The medium grey indicates that at least one of the twisted pairs was

[1]OCML (Operational Conceptual Modelling Language) is the knowledge modelling language provided in VITAL for detailed conceptual modelling or design.

ii) need to integrate with verification and validation - as noted by one of the reviewers, the link between tasks T231, T342 and task T232 (Verification and Validation) was unclear,

iii) end of project considerations - the need to make software robust, to take exploitation plans into account.

The influences of i) and ii) are described in sections 3 and 4. The influence of iii) was to produce two new visualizations. The first for the AC pilot. Obviously, this was necessary as this is one of the main VITAL workbench demos. The second visualization was for CLIPS. This was produced for the Open University's (OU's) own exploitation plan (before the work was halted). The two new visualizations are described in the following section.

2. NEW VISUALIZATIONS

2.1. The AC Pilot Visualization

The AC pilot is one of the main showpieces for the VITAL workbench. In addition to the fact that domain visualizations are required by the VITAL methodology, they are useful for explanations at exhibitions or conferences.

The domain of the AC pilot application is that of telephone line assignment. A customer requests a new telephone line with a number of services. The application allocates the cheapest line able to support the requested services. The application can also deal with other requests such as a change of address or the removal of a line.

Figure 1 below shows a screen snapshot from the AC pilot application.

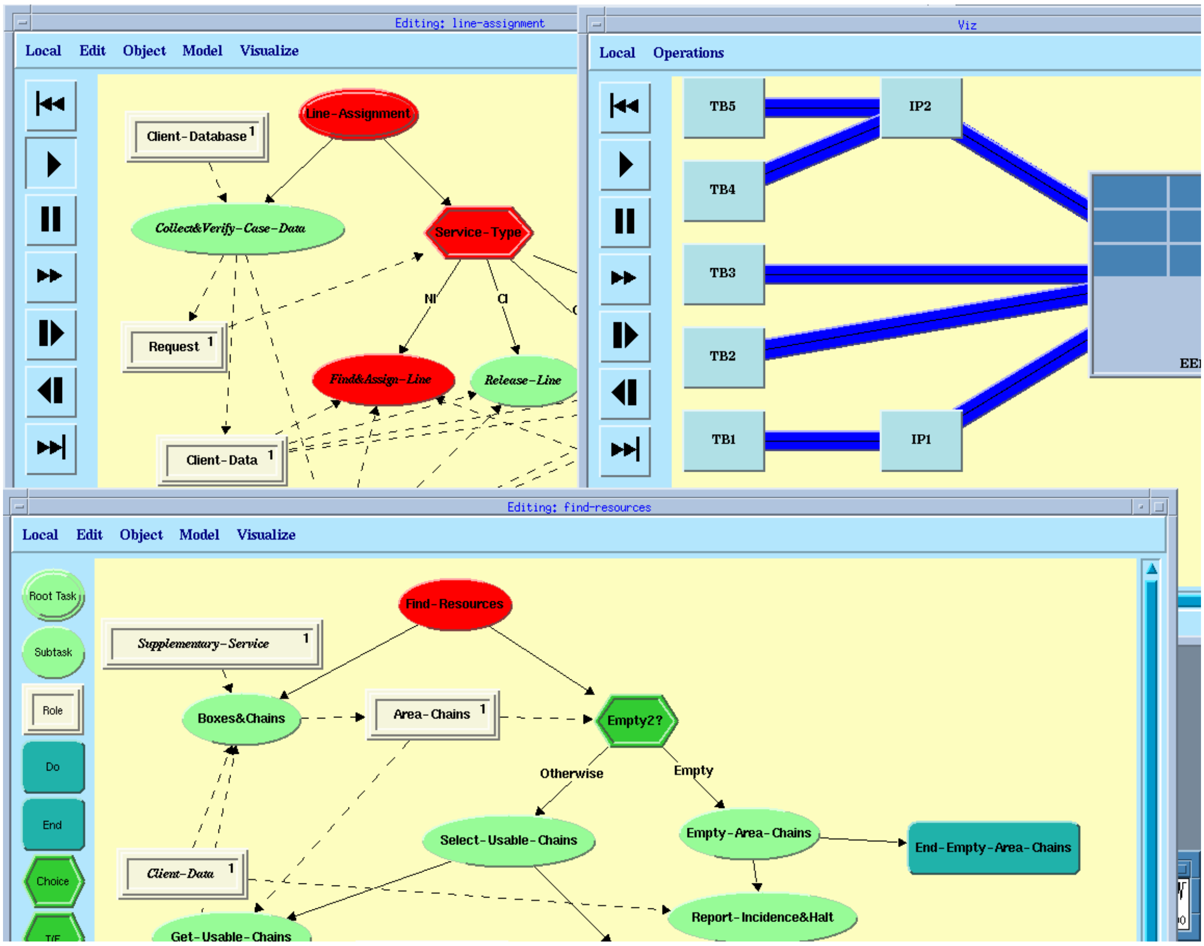

Figure 1. A screen snapshot from the AC pilot.

PREFACE

This task was stopped prematurely - see fax "Preperation (sic) of Vital PGM" (20th July) from Alain Rouge. This means that not everything was completed. The places where software components were not completed because of the stoppage are indicated within the text.

1. INTRODUCTION

The importance of debugging and validation within any software system is indisputable. "For every dollar you spend on software development you will spend two dollars on software maintenance" [Boehm, 1987].

This is the final report for the task T341 Bug Location Tool. It is the last of a trio of tasks aimed to provide debugging support within VITAL. The original objectives for these three tasks were (from the technical annex):

T231 Bug Location Methodology Objectives:

"Program debugging is a major part of any software development project and it is therefore important that it is addressed within VITAL. We shall develop a methodology for 'bug location', which is the task of tracing from the manifestation of a bug to its source."

T342 Visualization Tool Objectives

"The visualization tool will provide a set of techniques for visualizing the various models created by the VITAL tools."

T341 Bug location support prototype objectives

"The tool will enable knowledge engineers quickly to locate the source of bugs with various interrogation mechanisms."

A summary of the outcome and effort (in man-months) for the three tasks is given below.

Task	Effort (mm)	Outcome
Bug Location Methodology (T231)	17	A methodology for locating bugs within a KBS based on visualization support. A framework and methodology for constructing visualization systems called Viz.
Visualization Tool (T342)	14	An implementation of Viz. 7 visualization systems including Prolog, OPS5, the VITAL-KR, sisyphus I and II.
Bug Location Tool (341)	9	Two more visualizations: CLIPS (a public domain OPS5 like language) and the AC pilot application. A set of tools to aid 'inter-model' bug location and the construction of visualizations. A mapping to the relationship between this work and verification and validation. A protocol editor.

Even though this task was halted prematurely I believe that the trio of tasks have managed to satisfy the goals outlined nearly five years ago.

There have been 3 main influences on this task:

i) outcome of task T231 Bug Location Methodology - obviously the tool should follow from the methodology,

ESPRIT PROJECT P5365
VITAL
Task 3.4.1 BUG LOCATION TOOL

VITAL

A methodology - based workbench for KBS Life cycle support

Document Reference:
OU/DD341/D1.0 Date: 24/4/95

Activity:
T3.4.1 Status: Final Release

Distribution:
All VITAL Partners

Title:
The VITAL Bug Tool Report

Abstract:
In this document we describe the VITAL bug location tool. The last in a trio of tasks which provide the debugging support within VITAL

Authors:
John Domingue

IMPORTANT NOTE: 1 **Printing**

This document contains numerous colour screen snapshots and diagrams. In order for these to be legible this document MUST be printed with the **Colour/Greyscale** switch **ON**. This switch is a radio button in the print dialogue.

Collaborators (VITAL Partners):

SYSECA - SYSECA TEMPS REEL (Coordinator)
BULL - BULL CEDIAG
ONERA - ONERA
*OU - THE OPEN UNIVERSITY
UoH - THE UNIVERSITY OF HELSINKI
NOTT - UNIVERSITY OF NOTTINGHAM
AC - ANDERSEN CONSULTING
PTT - ROYAL PTT NEDERLAND NV
NOKIA - NOKIA RESEARCH CENTER

*marked partners are involved in this task