City Research Online

City St George's, University of London

Citation: Finkelstein, A., Kramer, J., Nuseibeh, B., Finkelstein, L. & Goedicke, M. (1992). Viewpoints: A Framework for Integrating Multiple Perspectives in System Development. International Journal of Software Engineering and Knowledge Engineering, 02(01), pp. 31-57. doi: 10.1142/s0218194092000038

This is the accepted version of the paper.

This version of the publication may differ from the final published version. To cite this item please consult the publisher's version.

Permanent repository link: https://openaccess.city.ac.uk/id/eprint/26496/

Link to published version: https://doi.org/10.1142/s0218194092000038

Copyright and Reuse: Copyright and Moral Rights remain with the author(s) and/or copyright holders. Copies of full items can be used for personal research or study, educational, or not-for-profit purposes without prior permission or charge, unless otherwise indicated, provided that the authors, title and full bibliographic details are credited, a hyperlink and/or URL is given for the original metadata page and the content is not changed in any way. For full details of reuse please refer to City Research Online policy.

City Research Online: http://openaccess.city.ac.uk/ publications@citystgeorges.ac.uk

Viewpoints: A Framework for Integrating Multiple Perspectives in System Development

A. Finkelsetin
J. Kramer
B. Nuseibeh
L. Finkelstein
M. Goedicke

(In) International Journal of Software Engineering and Knowledge Engineering 2(1):31-58, March 1992, World Scientific Publishing Co.

Viewpoints: a framework for integrating multiple perspectives in system development

A. Finkelstein, J. Kramer & B. Nuseibeh
Department of Computing, Imperial College, 180 Queens Gate, London SW7 2BZ
(acwf@doc.ic.ac.uk).

L. Finkelstein
Measurement and Instrumentation Centre and Engineering Design Centre, City University, London.

M. Goedicke
Fachbereich Informatik, Universität Essen, Essen.

0 Abstract

This paper outlines a framework which supports the use of multiple perspectives in system development, and provides a means for developing and applying systems design methods. The framework uses “viewpoints” to partition the system specification, the development method and the formal representations used to express the system specifications. This VOSE (viewpoint-oriented systems engineering) framework can be used to support the design of heterogeneous and composite systems. We illustrate the use of the framework with a small example drawn from composite system development and give an account of prototype automated tools based on the framework.

Key Words: composite and heterogeneous systems; perspectives; view; agents; specification method; CASE tools; software development environments; process modelling.

1 Introduction

The development of most large and complex systems necessarily involves many people - each with their own perspective on the system defined by their skills, responsibilities, knowledge and expertise. This is particularly true where the system is a composite system, that is one which deploys a variety of different technologies (software, hardware, mechanical and so on). Inevitably, the different perspectives of those involved in the process intersect and overlap, giving rise to a requirement for coordination. The intersections are, however, far from obvious because the knowledge within each perspective is represented in different ways. Further because development may be carried out concurrently by those involved, different perspectives may be at different stages of elaboration and may each be subject to different development strategies.

The problem of how to guide and organise development in this setting - many actors, sundry representation schemes, diverse domain knowledge, differing development strategies - we term “the multiple perspectives problem”.

This paper illustrates the multiple perspectives problem and introduces “viewpoints” as a framework for structuring, organising and managing these perspectives. The paper

describes viewpoints in some detail, defining the contents and use of their component parts. The application of viewpoints to method description and system specification is explained and tool support outlined. Some general conclusions are drawn.

2 Multiple Perspectives: an example

To show how our framework operates we shall use a simple composite system design problem - a lift system. It consists of a number of components (Figure 1).

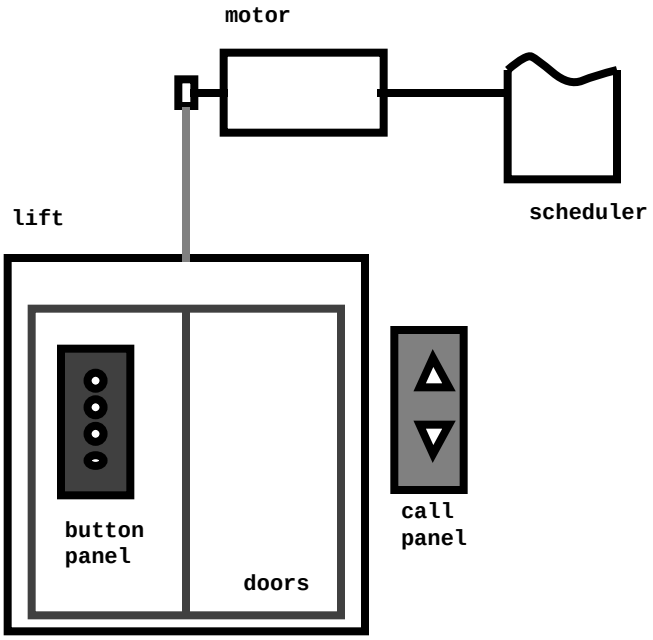

Figure 1: a simple lift system

Our design team is shown in Figure 2 with their responsibilities. Some team members are responsible for particular components of the system, Bob, for example, is responsible for the scheduler. Ken is not responsible for any particular component but rather for a global aspect of the system, its performance. Anne and Fred are jointly responsible for the motor. Joe is responsible for two components - the button panel and the call panel. Jane is responsible for the lift itself. Joe, in his capacity as button panel developer, and Tom, who is responsible for the doors, work for Jane.

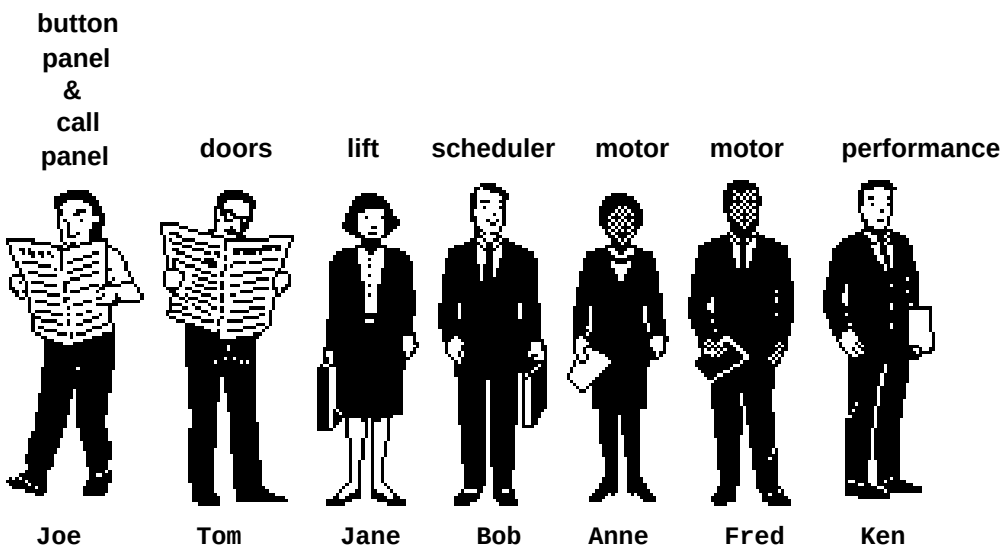

Figure 2: the design team

Clearly in our example the areas of responsibility overlap requiring team members to talk to each other: Bob (scheduler) must talk to Anne (motor) and Fred (motor); Bob (scheduler) must also talk to Jane (lift); Jane must talk to Joe (button panel) and Tom (doors), who must talk to each other; Ken (performance) must talk to everybody; and so on.

Different team members use different representation styles that are chosen to be particularly appropriate to their area of work (Figure 3). Joe, Tom and Jane use functional or part hierarchies to describe the button panel, call panel, doors and lift respectively. Bob uses action tables; Bob may in fact use many of these tables to model the scheduler. Bob also uses an object structure diagram. Anne and Fred both do system block diagrams of the motor. Ken uses a Petri net which is amenable to performance analysis.

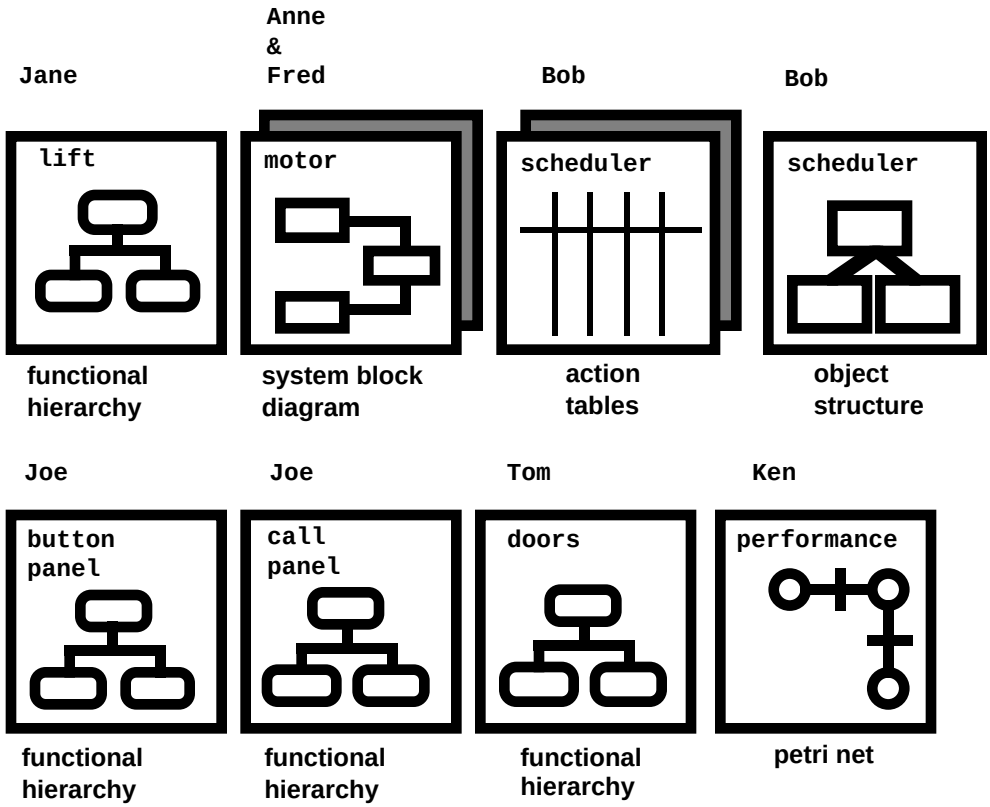

Figure 3: representation styles

Just to make things even more complex we will assume that Anne and Fred arrive at alternative motor designs as in Figure 4.

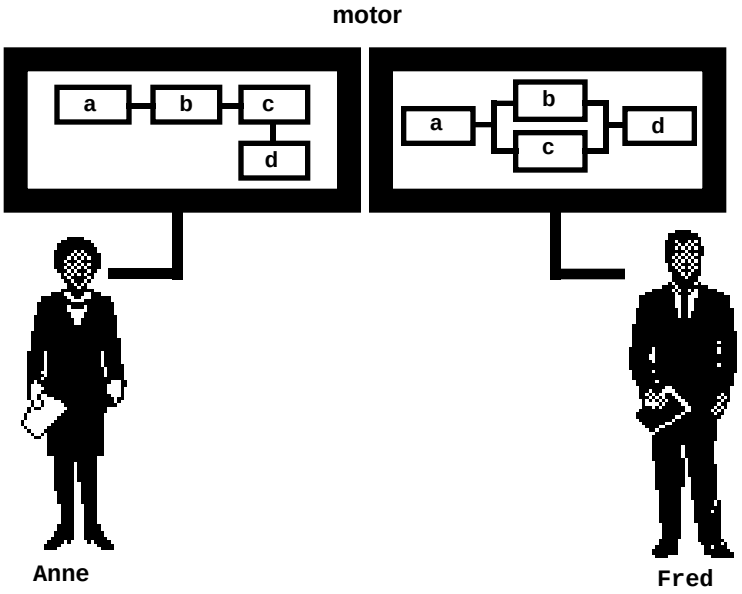

Figure 4: alternative designs

Joe and Tom, Figure 5, despite using the same representation scheme are using different development strategies. Joe is working bottom-up while Tom is working top-down.

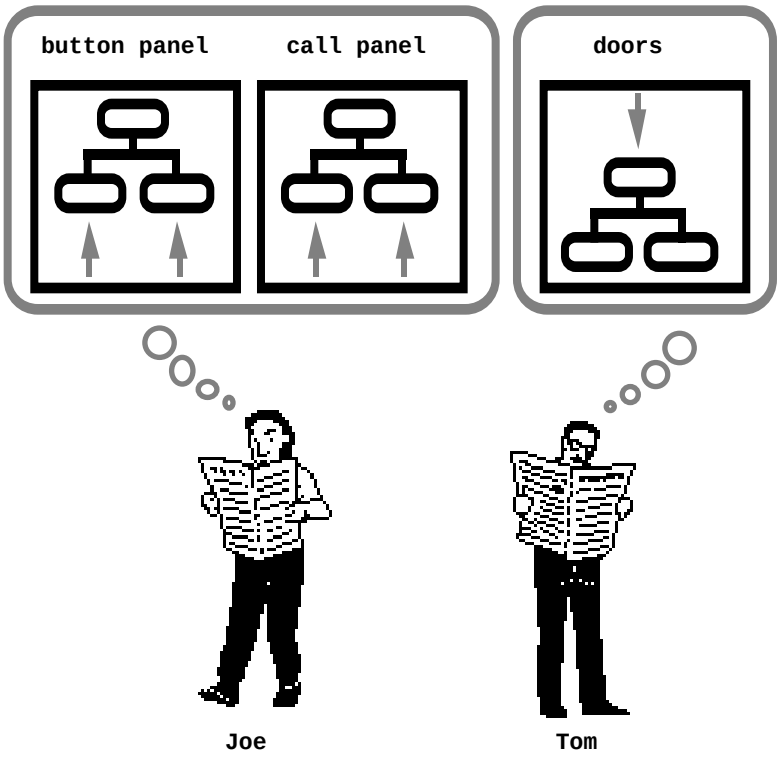

Figure 5: different development strategies

Finally, we introduce some “organisational structure” into the problem (Figure 6). Joe (button panel) and Tom (doors) must wait for Jane to finish before they start - her word is law. Anne and Fred (motor) can work concurrently with Bob (scheduler) - but still must liaise on matters of mutual concern.

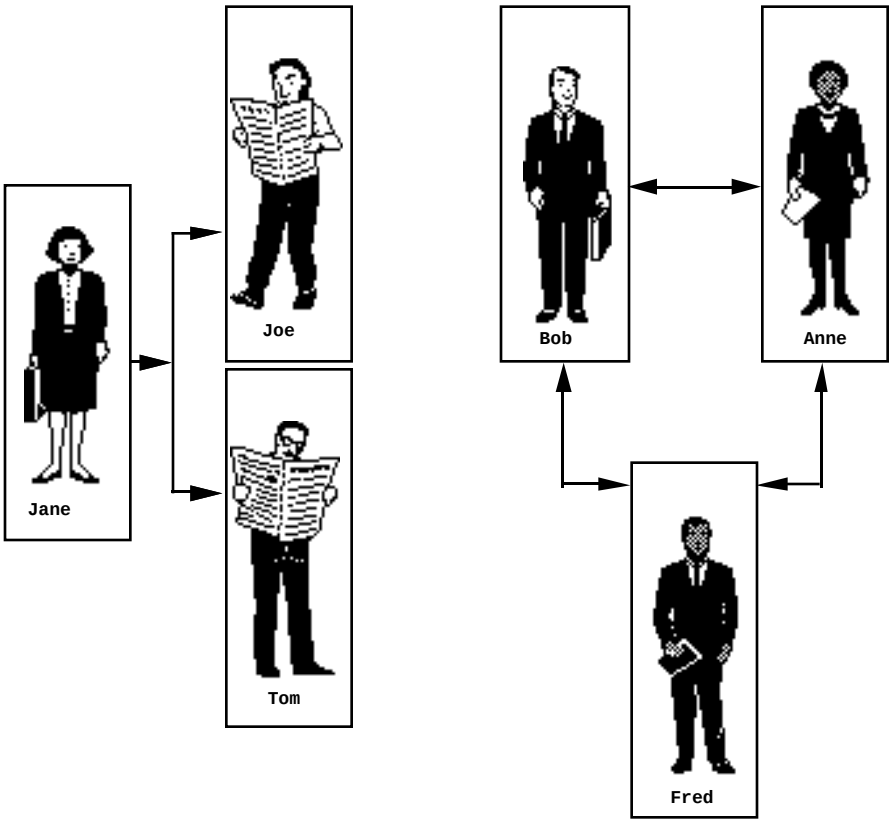

Figure 6: relation between team members

3 What is a Viewpoint?

The scene has been set for us to introduce our framework which will organise and support the example above. The basic building block of our framework is a viewpoint. A viewpoint can be thought of as a combination of the idea of a “actor”, “knowledge source”, “role” or “agent” in the development process and the idea of a “view” or “perspective” which an actor maintains. In software terms it is a loosely coupled, locally managed object which encapsulates partial knowledge about the system and domain, specified in a particular, suitable representation scheme, and partial knowledge of the process of design.

Each viewpoint is composed of the following components, which we call slots:

- a representation ***style***, the scheme and notation by which the viewpoint expresses what it can see;

- a ***domain***, which defines that part of the “world” delineated in the style;

- a ***specification***, the statements expressed in the viewpoint's style describing particular domains;

- a ***work plan***, describing the process by which the specification can be built;

- a ***work record***, an account of the history and current state of the development.

Before we look in any more detail at the content of each slot let us revisit our example.

Let us first consider Ken. A very simplified picture of Ken's viewpoint is given below, Figure 7. The representation style Ken uses is Petri Nets and his domain of concern is Lift System Performance (in Ken's case the domain is not tied to a specific physical component). The specification is Ken's current knowledge about lift system performance expressed as a Petri Net (a small example of such a specification is given as an illustration). The work plan explains how to build a Petri Net and how, and in what circumstances, to check consistency with the other viewpoints. The work record gives the current state of Ken's specification and an account of its development in terms of the work plan.

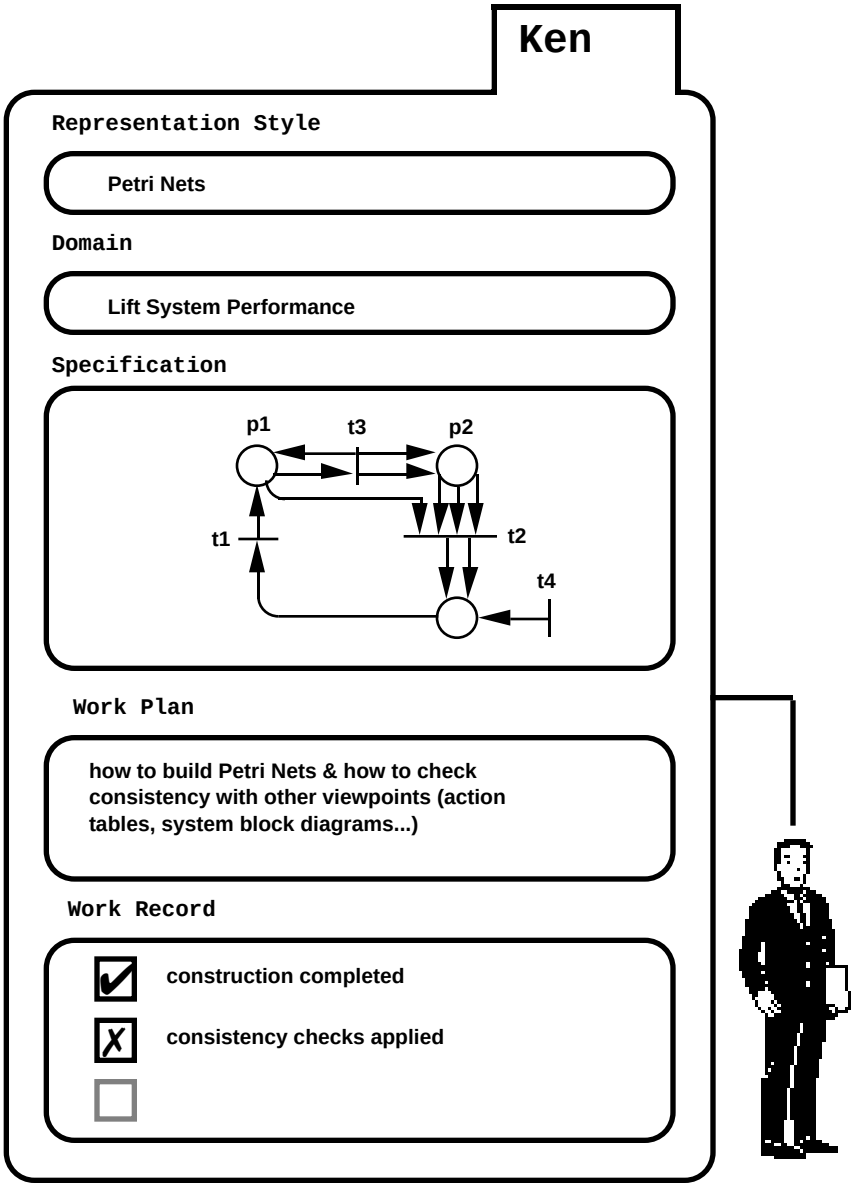

Figure 7: a simple viewpoint

Though thinking of Ken's responsibilities and knowledge as constituting a single viewpoint is straightforward it is not always possible to associate a physical agent with a viewpoint. Joe is a case in point, he is responsible for both the button and the call panel. In

our framework we treat this as, in effect, two viewpoints “Joe: button panel” and “Joe: call panel” each with different domains of concern (Figure 8).

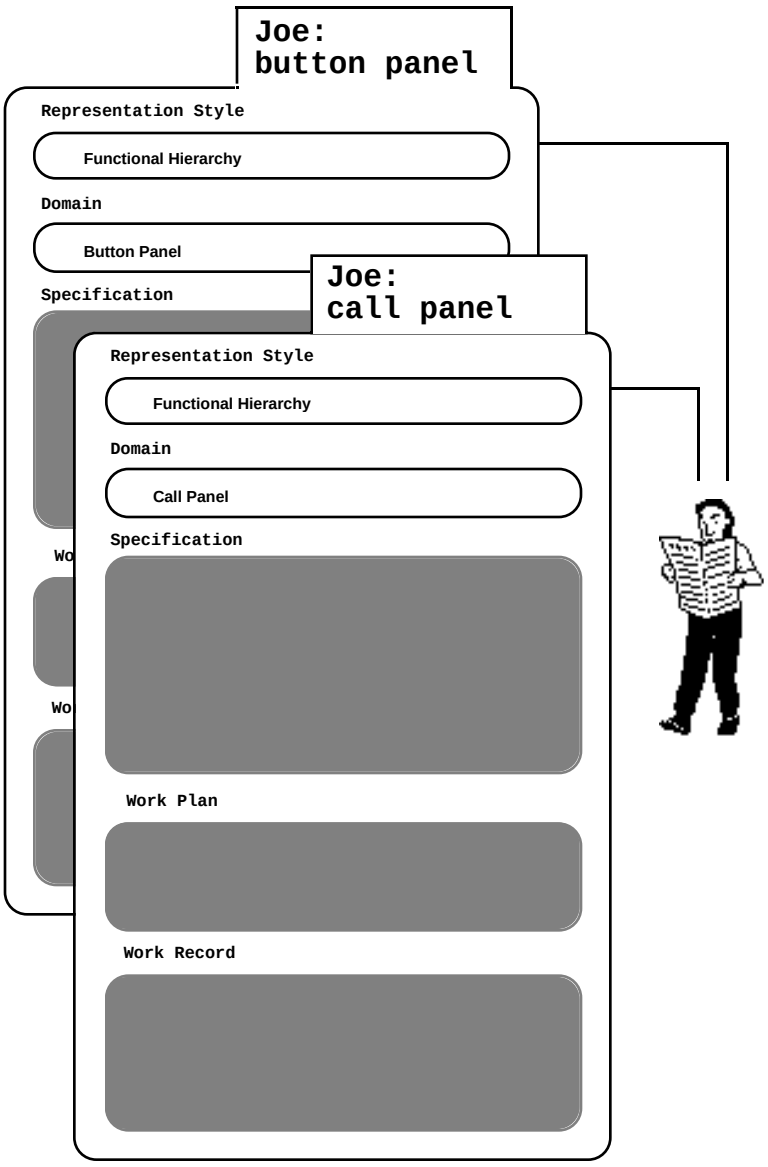

Figure 8: different viewpoints for different domains

Bob is using action tables as a representational style. He requires several action tables to build a description of the scheduler. Action tables are a partitioned representation style - one action table is used for each information transforming entity in the system. In our framework each table is given a separate viewpoint, reinforcing the separation of concerns provided by the representation style. Each viewpoint has a different domain - in the case of Bob some examples might be the request manager (handling incoming service requests) and the priority manager (assigning services to those requests), see Figure 9.

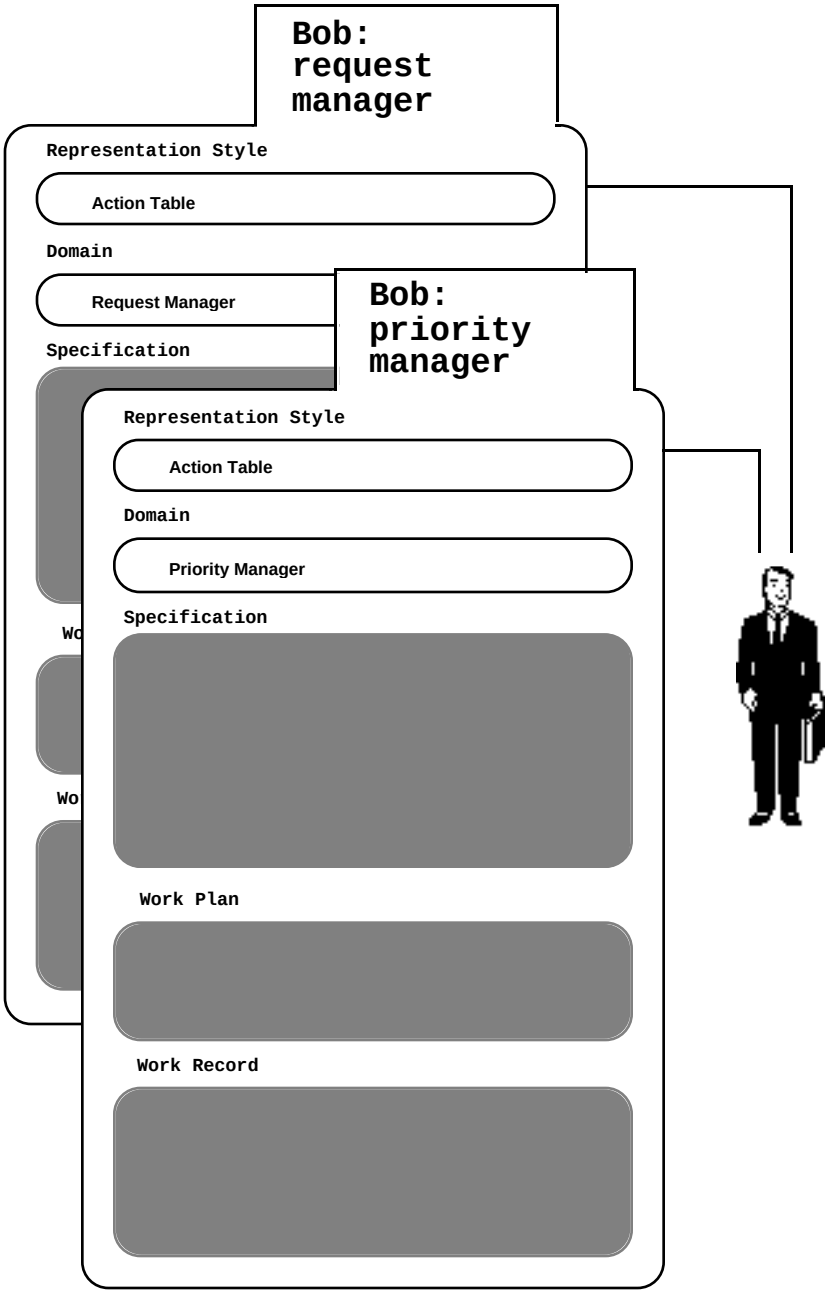

Figure 9: different viewpoints in a structured representation style

Bob also has a separate viewpoint for the object structure diagram. The domain of that viewpoint is the scheduler and its immediate environment.

To capture the distinction between the viewpoint and the person who is responsible for it - Bob, Anne, Tom, Jane or whoever - we use the term “owner”.

Figure 10 shows the collected viewpoints for the lift system (for simplicity we have not listed all of Bob's action table viewpoints).

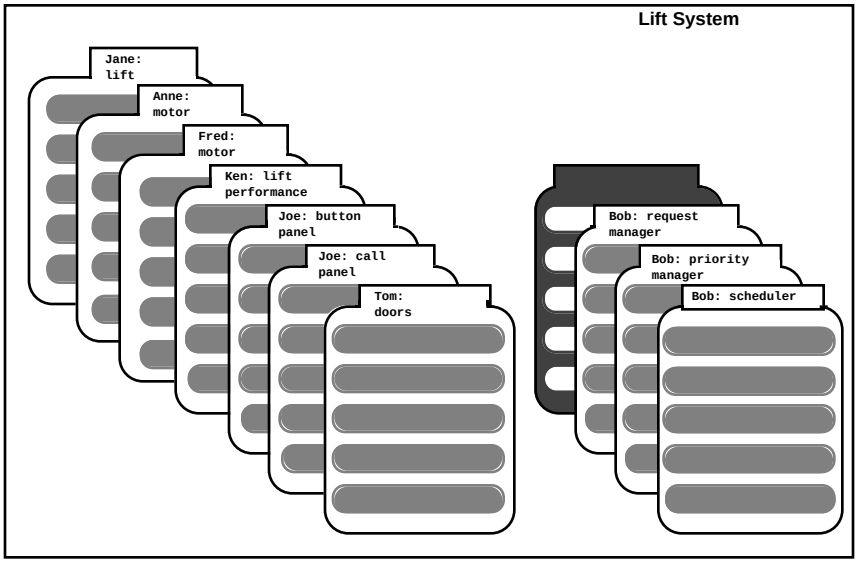

Figure 10: lift system viewpoint configuration

4 What is in a Viewpoint?

So far we have only sketched the content of each of the slots of the viewpoint. Now we will look at them in more detail.

4.1 Representation Style

The representation style slot (essentially the representation language definition) is composed of two parts:

- ***objects***, the elements of the representation style;

- ***relations***, relations that hold between objects.

Figure 11 shows the simple case of Jane who uses a functional hierarchy as her representation style. The primary linguistic element of the style is the functional_element, graphically shown as an ellipse. Each functional_element has a name. Functional elements are related by the part-of relation, graphically shown as a vertical line relating a super-ordinate functional_element to a sub-ordinate functional_element.

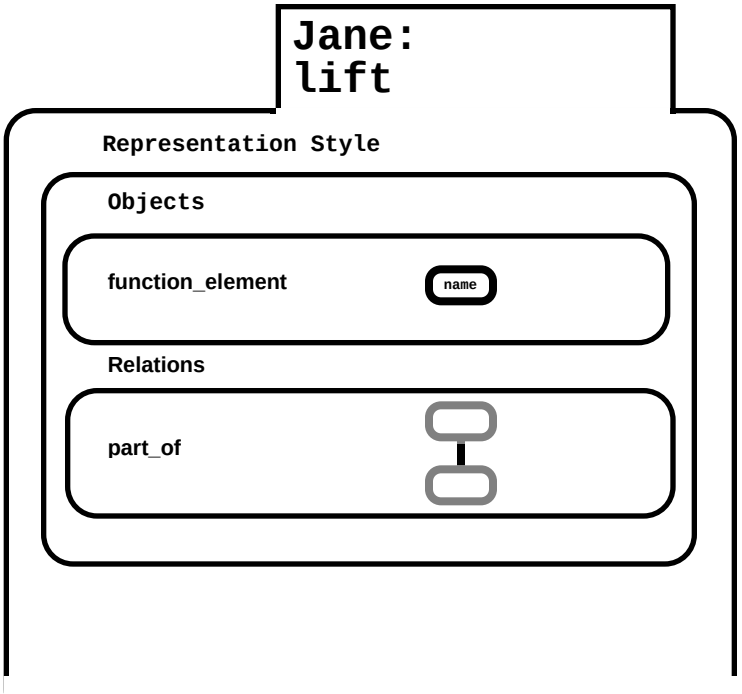

Figure 11: representation style slot

4.2 Domain

The content of the domain slot is the name given to that part of the “world” seen by the viewpoint. Examples have been given above in Figures 7, 8 and 9.

4.3 Specification

Given the representation style used by Jane a specification built by her might look as shown in Figure 12.

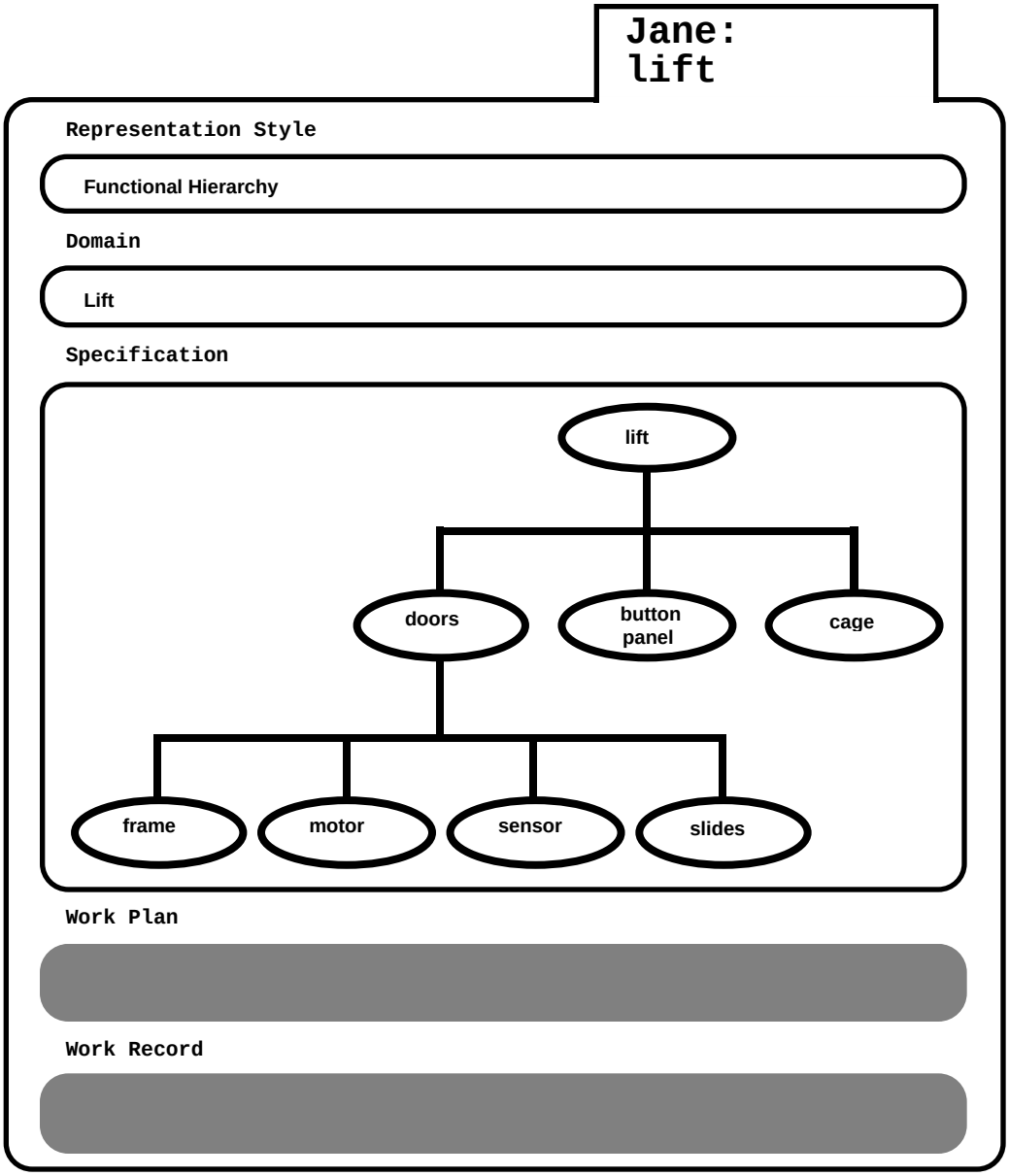

Figure 12: specification slot

4.4 Work Plan

The most complex part of the viewpoint is the work plan. We divide the work plan into four parts, they are:

- ***assembly actions***, which contains the actions available to the developer to build a specification;

- ***check actions***, which contains the actions available to the developer to check the consistency of the specification;

- ***viewpoint actions***, which create new viewpoints as development proceeds;

- ***guide actions***, which provide the developer with guidance on what to do and when.

Using our example we shall examine each of these in turn.

Figure 13 shows the assembly actions available to Jane - she can add and relate objects (or remove objects and relationships). These are all she needs to build a specification, though not necessarily a consistent specification.

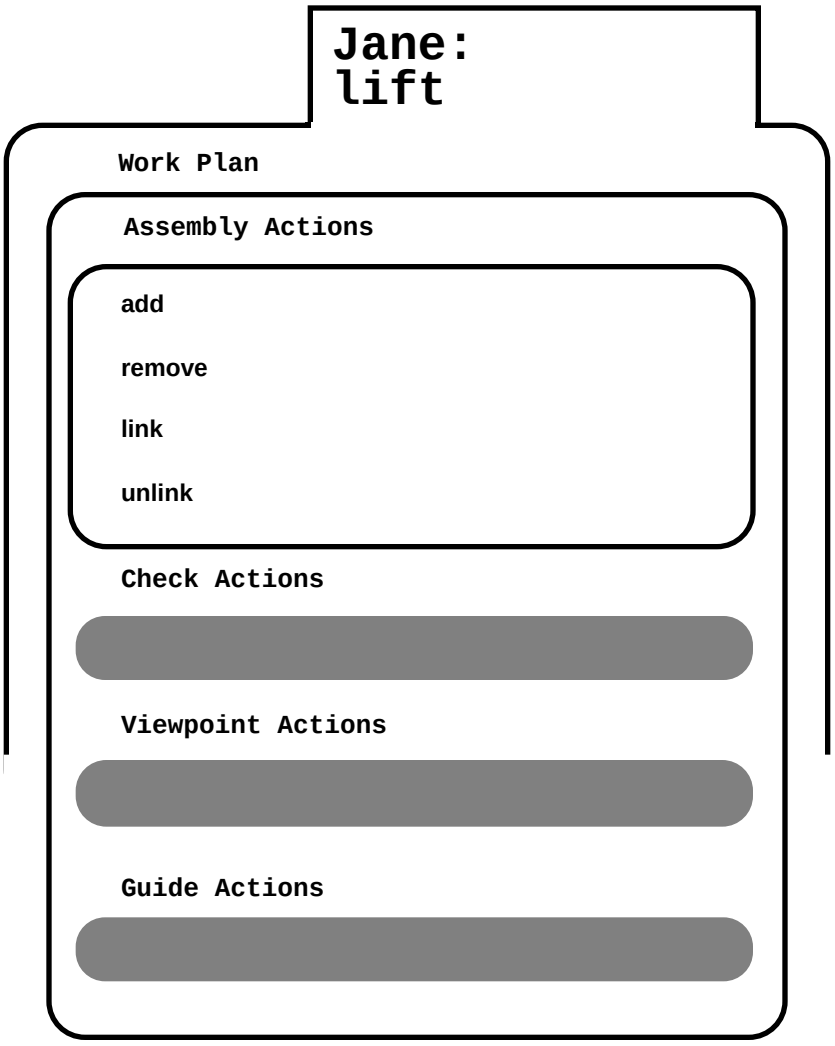

Figure 13: assembly actions

Check actions are the core of the viewpoint framework.

Check actions belong to one of two groups eac with a different scope:

- ***in-viewpoint checks***, check the consistency of the specification within the viewpoint where the check action has been invoked;

- ***inter-viewpoint checks***, check the consistency of the specification with those maintained by other viewpoints.

We will illustrate each of these with some examples. Figure 14 shows a specification prepared by Jane. Jane has placed a functional_element named doors directly below that of another functional_element identically named. Doing this has created an inconsistency

within her viewpoint, clearly doors cannot be part_of doors. This inconsistency would be picked up by an in-viewpoint check. Jane should, at some appropriate moment, be required to fix the inconsistency, perhaps by changing the name of the lower element and repeating the check action that pointed to the inconsistency.

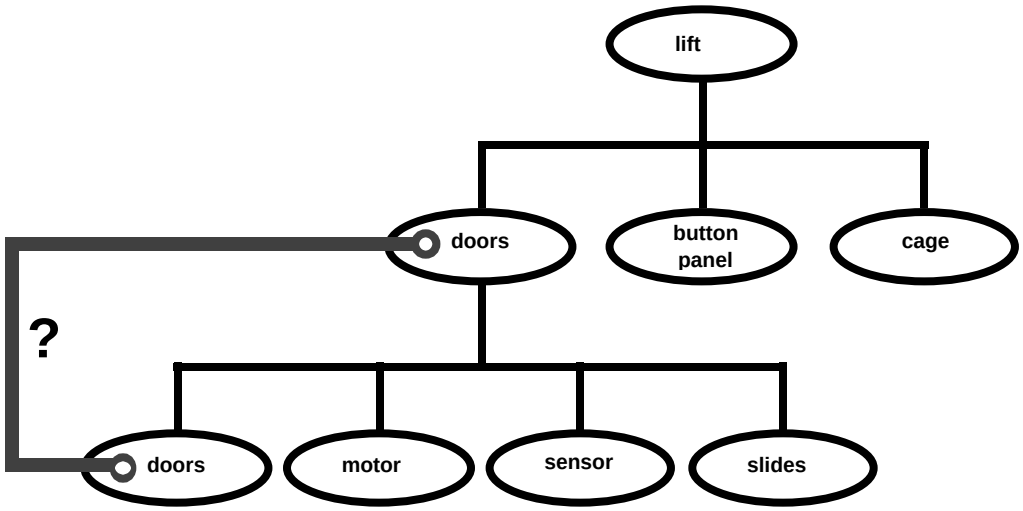

Figure 14: in-viewpoint check

Figure 15 shows two viewpoints, “Jane: lift” and “Tom: doors” which share a common template. The viewpoints are partially overlapping. Jane has gone some way towards specifying the doors. She has broken down the doors into parts, Tom needs to provide more detail. Checking the consistency of the overlapping part of the specification by means of an inter-viewpoint check action can take two forms. The first, and most obvious, is where Jane and Tom proceed to specify the doors independently. In this case they each arrive at a specification and check their consistency with each other, as in Figure 15. Almost inevitably some inconsistency will arise, in our example between the use of the functional_elements ‘slides’ and ‘base’, which Jane and Tom will have to fight out.

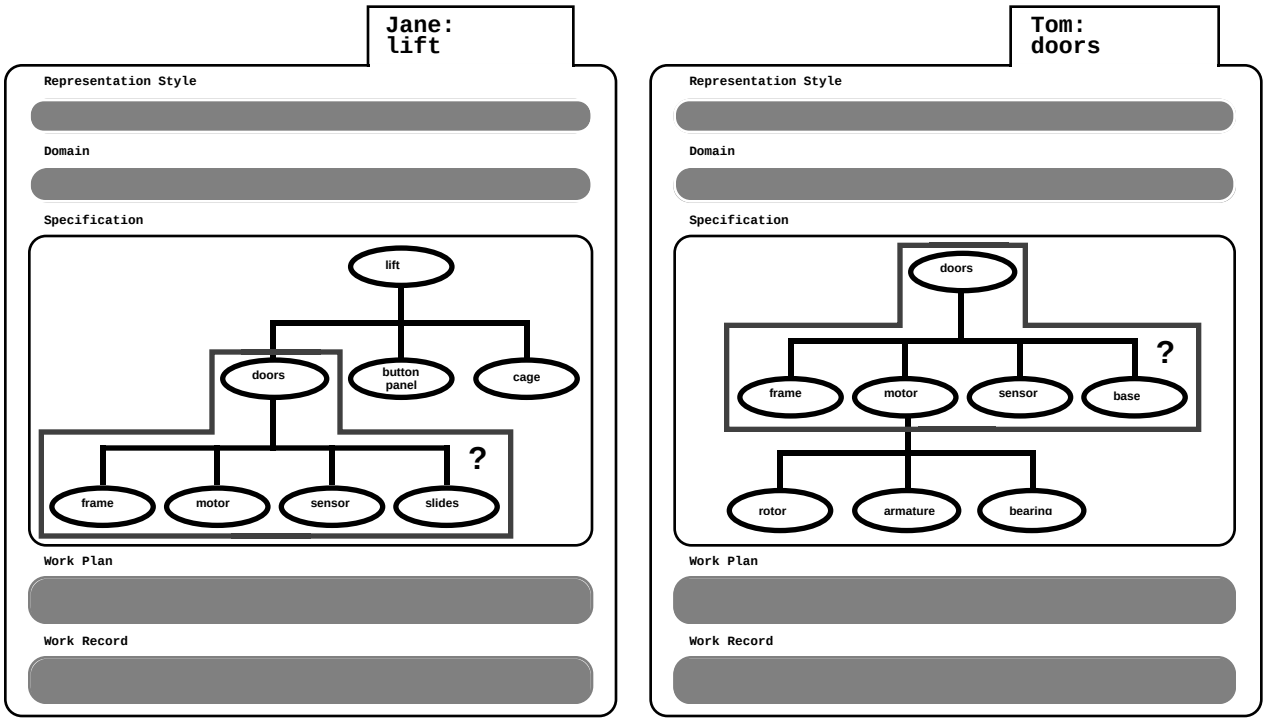

Figure 15: inter-viewpoint check ("resolve")

The second case, and the one we anticipated in section 2, is where Tom is subordinate to Jane. In this case, as in Figure 16 Jane completes the relevant part of the specification and simply transfers it to Tom's blank viewpoint, Tom then elaborates it. The distinction between these two forms of check can be thought of as similar to that in a Prolog query which can be used to check the consistency of some facts, returning true or false, or the facts satisfying some general rule. We call these two forms of check action application "resolve" and "transfer" respectively.

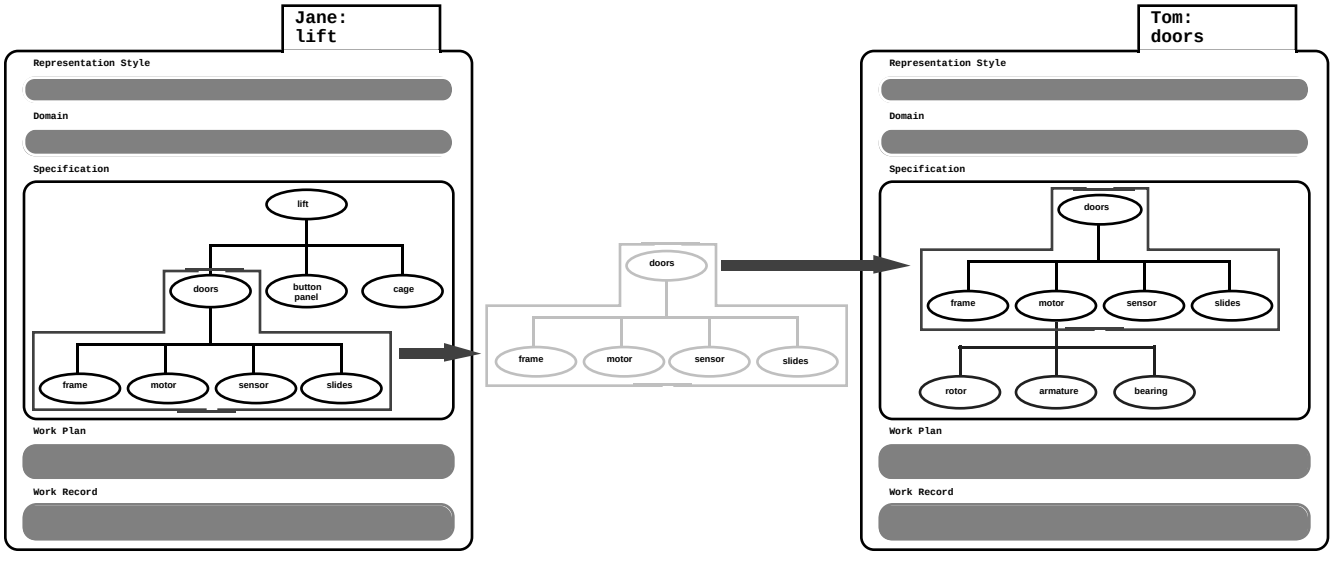

Figure 16: inter-viewpoint check ("transfer")

Of course, an inter-viewpoint check may also be applied where the two viewpoints overlap but the representation styles are different. In Figure 17 we show a simple example of this.

There is an overlap between two viewpoints, one using action tables and one using object structuring to express aspects of the scheduler (coincidentally they are both Bob's), "Bob: request manager" and "Bob: scheduler". Object structuring diagrams identify the main information processing elements of the system and their relation to each other. Action tables show the flow of information through each element, namely, the input information; the object which is its source; the transformation that is performed on it; the output information and the object which is its destination. It is important that the sources (and the destinations) used in the action table correspond to the objects identified in the object structure diagram. An inconsistency between the two suggests something seriously amiss.

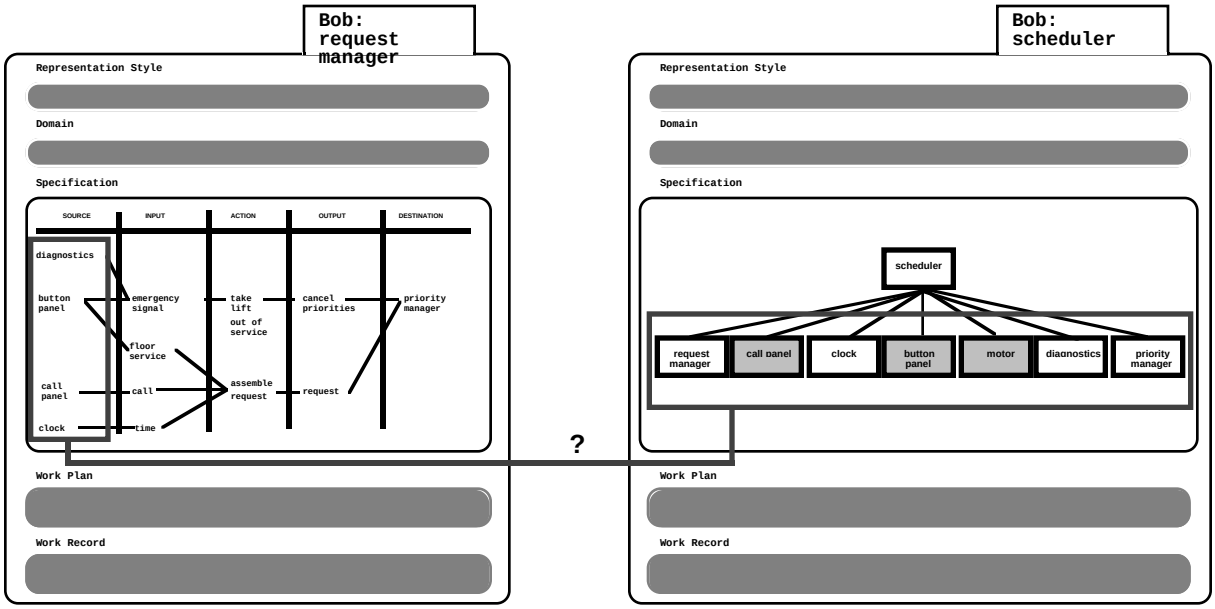

Figure 17: inter-viewpoint check with different representation styles

A less obvious example is given in Figure 18. Bob's action tables and Ken's Petri net model of performance overlap. We suggest a partial relation between the actions identified by Bob and the transitions used by Ken.

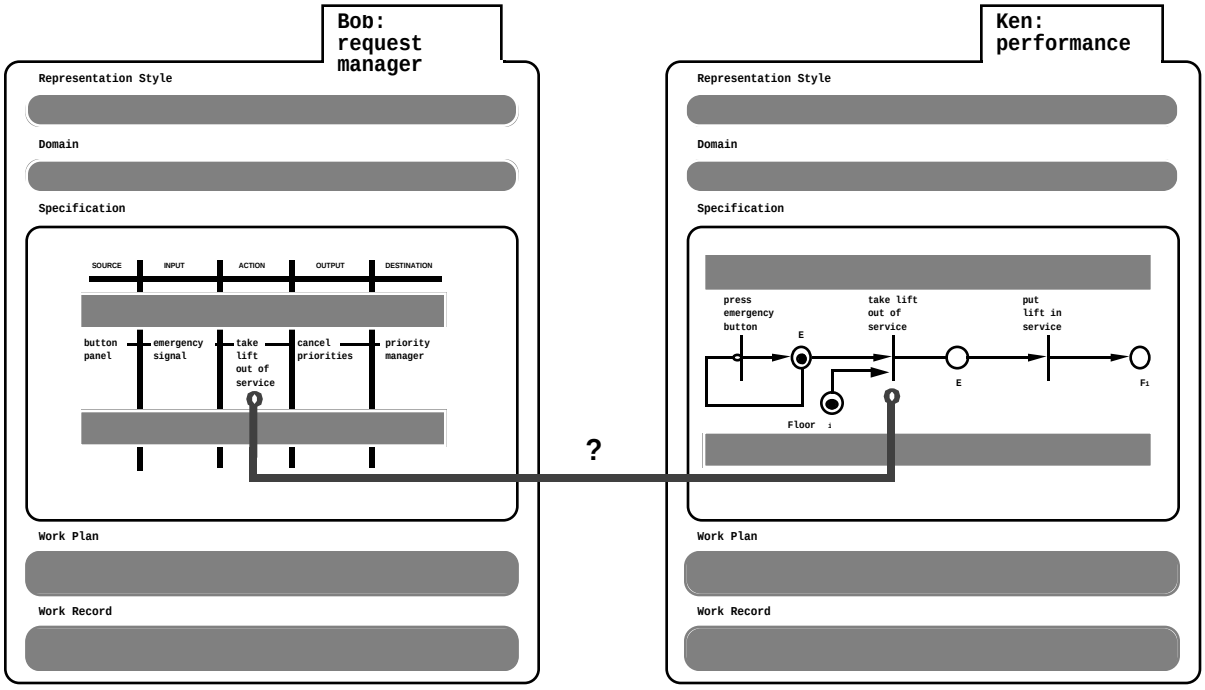

Figure 18: inter-viewpoint check with different representation styles

At this point it is worth recalling “Anne: motor” and “Fred: motor”. They each maintain an alternative version of the same domain in the same representation scheme. Pointing to the inconsistencies between these versions is simply a special case of an inter-viewpoint check.

Viewpoint actions provide the ability to create new viewpoints as development proceeds. To understand them we need to add a further concept - that of a viewpoint template. A viewpoint template provides a kind of viewpoint “type” from which new viewpoint instances can be created. It is simply a viewpoint in which the specification, domain and work record slot are empty. From our example it is clear that many of the viewpoints share a template: “Tom: doors” and “Jane: lift” (note that Joe and Tom who share representation styles do not have a common template because they use different development strategies); “Anne: motor” and “Fred: motor”; “Bob: request manager”, “Bob: priority manager”; and so on. Figure 19 shows Bob's action table viewpoints alongside an action table template.

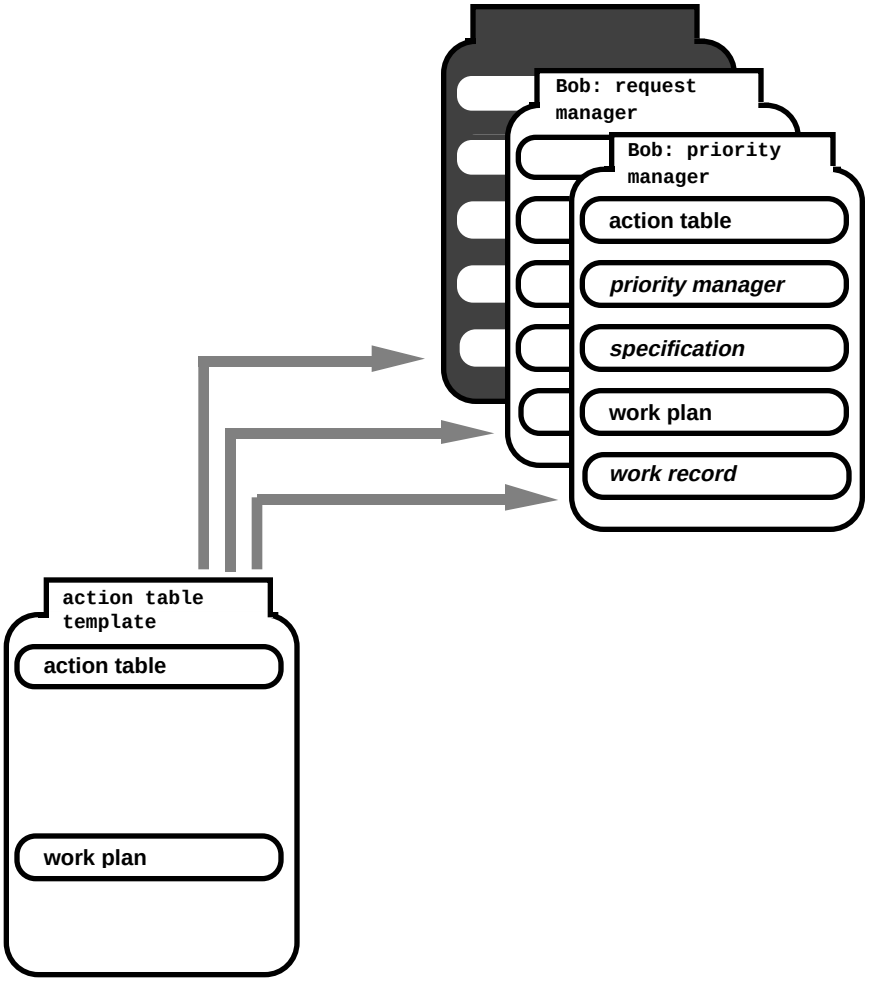

Figure 19: a viewpoint template

New viewpoints are dynamically created from the appropriate template by the viewpoint actions. Again, this is easier to see from an example. Bob needs an action table for each of the information processing elements of the scheduler - these are identified in the object structure diagram as unshaded boxes (Figure 20). Having completed the object structure diagram for the scheduler he can create blank action table viewpoints, new instances of the action table template, for each of those elements. They would then be elaborated in turn. In a similar fashion the completion of “Jane: lift” would create a blank viewpoint for “Tom: doors” followed by the information transfer discussed above.

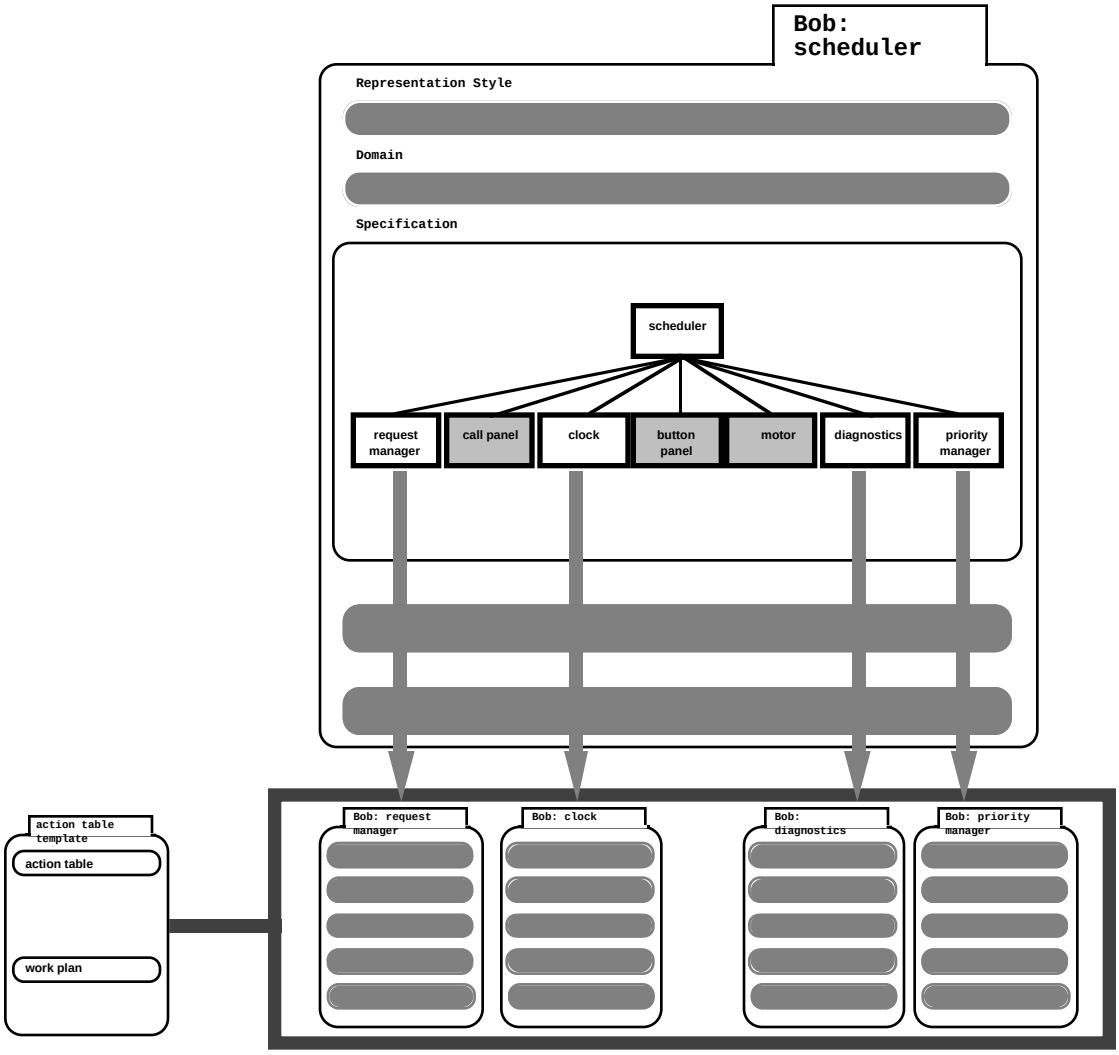

Figure 20: creating new viewpoints

The last class of actions in the work plan are guide actions. Up to this point we have only discussed what a viewpoint *might* do - assemble its specification, check consistency, spawn other viewpoints - not what it *ought* to do. This section of the work plan contains a formal description of the development strategy to be adopted. It sets out when certain check actions and viewpoint actions are permitted and when they are obliged to be performed. It is in effect a "process program". Developing the specification from within the viewpoint can be thought of as "enacting" this model of the development. Thus for example: Bob is permitted to perform an in-viewpoint check at any time but is obliged to perform all such checks and achieve an internally consistent specification immediately prior to performing any inter-viewpoint check; all available inter-viewpoint checks must be performed prior to the performance of a viewpoint action.

4.5 Work Record

The work record contains a development history expressed in terms of the work plan. It provides, in effect, a rationale for the current state of the specification. It details the actions which have been performed. For example it would show Jane's performance of the in-viewpoint check illustrated in Figure 14, the assembly actions (unlink, remove and so on) performed to regain internal consistency and the repetition of the relevant checks.

4.6 Definition

So far in this paper we have not discussed how exactly the representation style (objects and relations), the work plan (assembly actions, check actions, viewpoint actions and guide actions) and the work record are defined. We tend to favour heterogeneity in this as in other things and our framework does not restrict the manner in which this is done. In our current work we have used a simple Prolog-like scheme which has the advantage of being relatively easily implementable. Alternative schemes are under investigation.

5 Using Viewpoint Templates to Describe Methods

We have introduced viewpoints by looking at a specific example, the lift system, and seen how a particular configuration of viewpoints (Figure 10) is developed. In other words, we have looked at viewpoints from the standpoint of the system developer or "method user". In the example the particular set of representation styles and work plans that we selected were arbitrary, or rather they were chosen to illustrate the viewpoint concept. Of course, many real developments are done in accordance with a method, like HOOD, CORE, SSADM, SADT and so on, which prescribe the representation styles and associated work plan. The viewpoint framework provides us with a means of describing such methods - as configurations of viewpoint templates. In the discussion below we look at viewpoints from the standpoint of the "method designer".

Figure 21 shows how this might work. We show the configuration of viewpoint templates that might be used to support Jackson System Development (JSD) (Jackson 1983), a prime example of a system development method. JSD, like most system development methods has many simple (and predominantly graphical), highly redundant representation styles. It provides a work plan and many consistency checks, for example of the relation between entity lists and structure diagrams or structure diagrams and structure text.

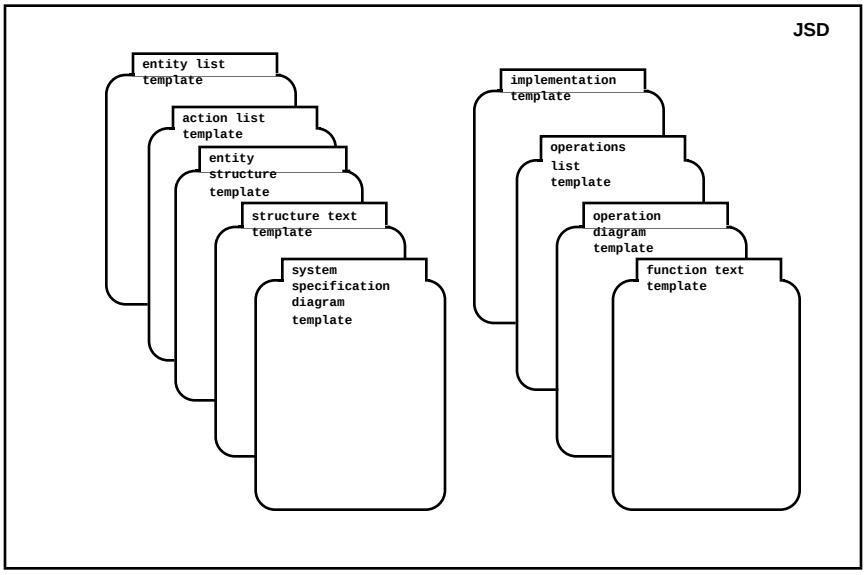

Figure 21: a configuration of viewpoint templates for JSD

In describing an existing method in this manner some judgements have to be made. For example in JSD how the actions of each entity are ordered in time is described in structure text. This same representation is subsequently enriched, during the function step, by the addition of operations to its structure. Should we treat the enriched representation as a separate, distinct, viewpoint template or use a more complex structure text viewpoint template? Though JSD treats each separately in different “steps” an argument could be made for adopting a single viewpoint template - the division is simply an artefact of the original textual, and hence sequential, account given of the method. We prefer using multiple templates both to retain, where possible, a link to the original step-based structure of the method, but also because the addition of functional information seems to introduce a genuinely new perspective into system development (by contrast, for example, it would not be appropriate to introduce a further viewpoint template for the refined entity list produced after entity structuring).

Another example of the judgements that must be made in describing an existing method is determining how (and indeed whether) to exploit the structuring provided by the representation style. In other words how big the specification contained by a viewpoint ought to be. A good example of this is the action list which contains a list of actions with accompanying textual descriptions. We could write our template in such a way that each item on the list (action and description) has its own viewpoint, alternatively we could use a single viewpoint to capture the whole list. Opinions may well differ on whether the extra structuring is worth the overhead of using the full power of the framework, this will at least be partly dependant on the size and complexity of the problems to be tackled.

The same approach which we applied to JSD can be applied to methods which use formal representation schemes. Complex formal specification languages can often usefully be broken down into many viewpoints with simple checks to manage consistency between the language fragments. Applying the framework in a more formal setting does however point to the fact that though there are many formal specification languages, there are very few formal methods which give direction on how a specification in that language is to be built.

It should be noted that the ability to use many different representation styles in conjunction does not necessarily make it either sensible or easy to do so. Good methods should be composed of viewpoint templates which complement each other and should be organised so as to allow development by incremental completion. The relation between any two representation styles may be extremely difficult to determine (for example CSP and Z). The same applies between two components in complex structured styles. This does not mean that the viewpoint framework cannot be used, there are usually many simple textual and knowledge management, “coarse-grain”, checks which are useful and “cheap” to provide. It is certainly not necessary (or possible) to guarantee full consistency - it is often the case that a viewpoint is valuable, irrespective of its consistency with others. A good example of this might be certain non-functional requirements.

Many different methods have techniques in common, think for example of the 1001 methods using data flow diagrams for some variant of structured analysis. Because viewpoints provide an organised way of setting down representation style and work plan it should be possible to reuse templates from method to method. Clearly these templates will have to be

tailored for their new context, notably the inter viewpoint (inter template) checks and the viewpoint actions. However, the scope of the changes should be fairly restricted.

6. Tool Support

Viewpoint Oriented Systems Engineering (VOSE) offers ample scope for tool support. On one hand, tools are required to facilitate method development, description and integration. On the other hand, tools are also required to support method use within the VOSE framework. This includes tool support for individual viewpoint development, as well as project management support for navigating around viewpoint configurations.

The √iewer (Figure 22) is a prototype environment supporting the VOSE framework. Implemented in Objectworks/Smalltalk V4.0, it is intended to support method development and integration, to facilitate the construction of individual viewpoint tool support, and to allow the smooth transition to, and manipulation and management of, viewpoints during method the use phase.

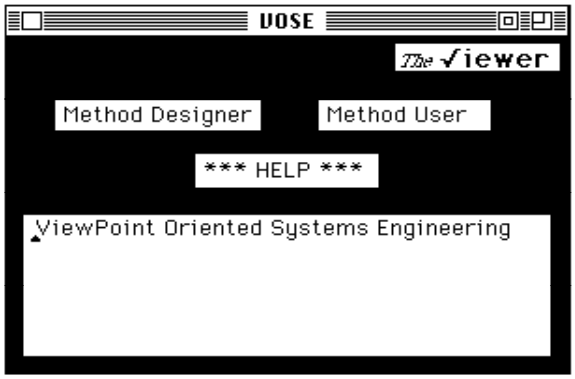

Figure 22: The √iewer

The Method Designer button takes the user through the method development and integration path, where tools are provided for describing and collecting viewpoint templates. These tools are accessed through template browsers, which provide the capabilities for textual and graphical descriptions of objects and relations in the representation style slot, and development actions and rules in the work plan slot. Figure 23 shows a typical template browser window.

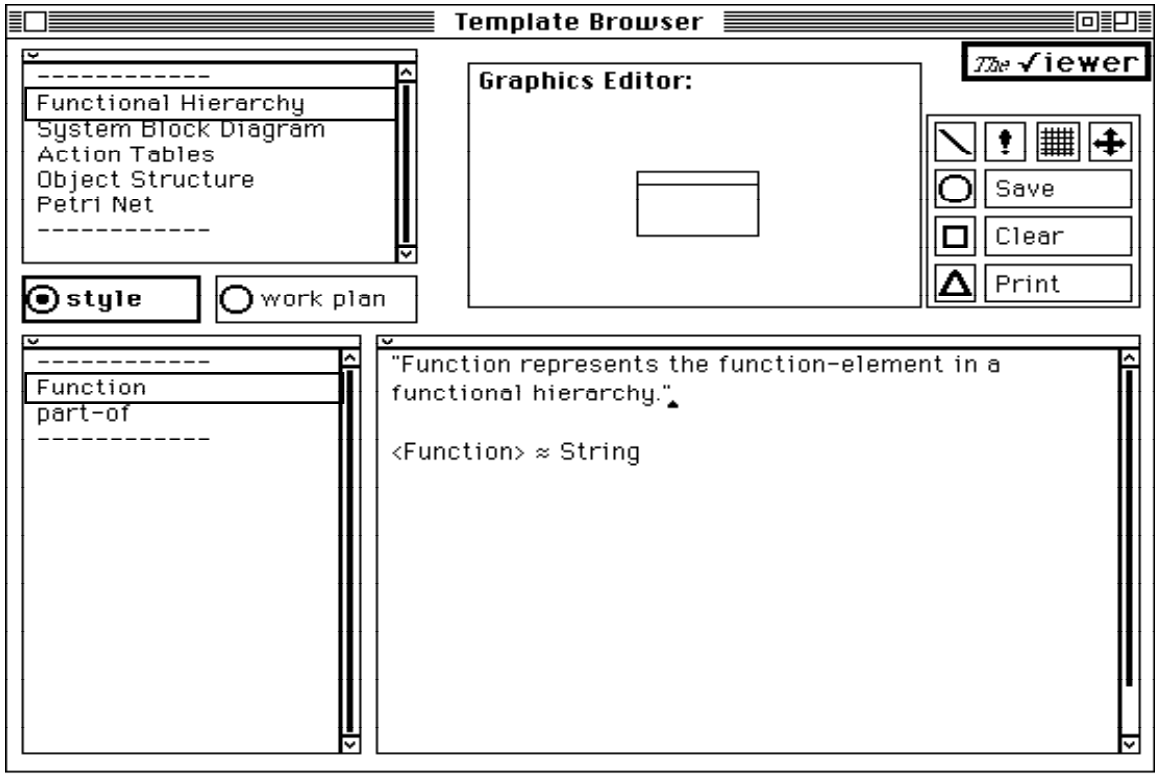

Figure 23: a viewpoint template browser window.

The top-left window pane lists the viewpoint templates currently being browsed, with the selected template highlighted. The "style" and "work plan" buttons are mutually exclusive switches which are selected using the mouse button. The bottom-left window pane lists the components of the style or work plan for the selected viewpoint template. In this example, the single object "Function" and relation "part-of" are listed. The two right-hand window panes elaborate the graphical (top) and textual (bottom) descriptions of the selected style or work plan components. Thus, in this example, the graphical shape of a "Function" is displayed (and may be edited using the drawing tools provided), and its type definition and explanatory comments are shown below.

The user can also follow the method use and management path. Here tools are provided for monitoring, inspecting and/or editing individual viewpoints or their configurations. More specifically, users are initially provided with a viewpoint configuration browser (Figure 24) in which one or more projects may be created and any number of viewpoints instantiated for each project. A viewpoint configuration table shows all templates and domains for which viewpoints have been instantiated.

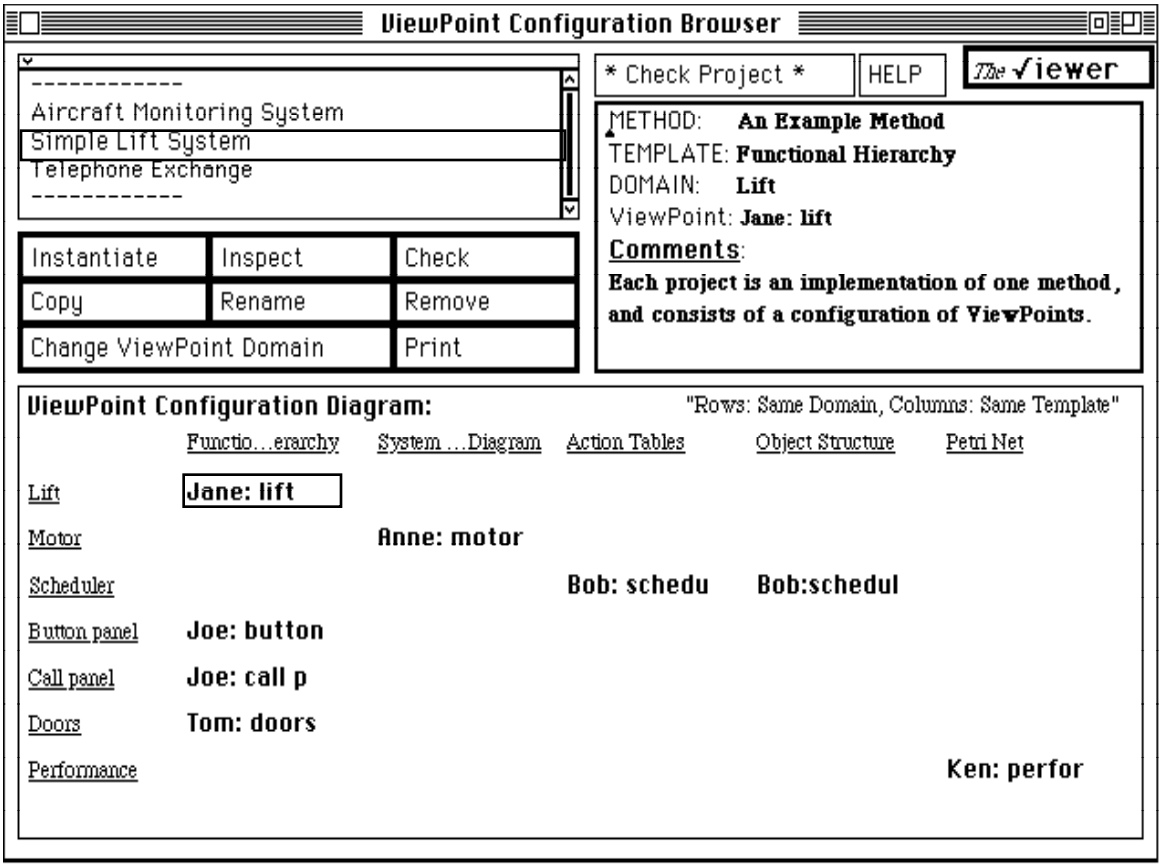

Figure 24: a viewpoint configuration browser window.

The top-left window pane lists the projects that have been created, with currently selected project highlighted. For any selected project, the corresponding viewpoint configuration table is displayed in the bottom window pane. The rows represent viewpoints of the same domain, while the columns represent viewpoints instantiated from the same template. Once a viewpoint has been selected from the viewpoint configuration diagram (and consequently highlighted), the panel of buttons above the diagram may be used to manipulate the selected viewpoint. Clicking the 'Inspect' button on the browser produces a viewpoint inspector for the selected viewpoint

Viewpoint inspectors provide the workspace and tools to perform actual specification development. Development activities and state are logged on to the work record. From within this inspector the assembly check and guide actions defined in the viewpoint template may be performed.

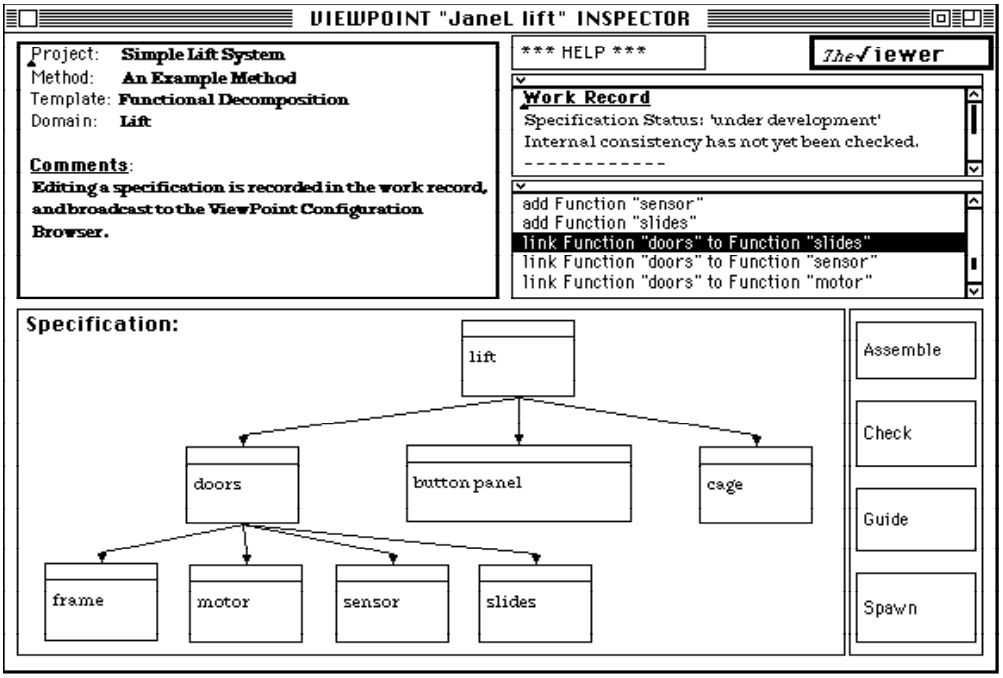

Figure 25: a viewpoint inspector window.

The bottom pane displays the viewpoint specification created by the 'Assemble' commands which pop-up when the 'Assemble' button is clicked. The work record, shown in the top-right window panes maintains a record of the specification development history, with any design rationale and/or annotations entered by the user or automatically generated by the system.

Checking for consistency requires the selection of the scope (in- or inter -viewpoint) and mode (resolve or transfer) of application of development rules. Method guidance is provided via the 'guide' button. These are as yet only partially implemented.

Several different viewpoint inspectors may be opened at any one time, and we can thus envisage distributed systems development with several viewpoints being developed on different nodes in a distributed environment. Such development may be monitored by a project manager using a viewpoint configuration browser.

7 Conclusions

In the account above we have shown how the viewpoint framework can be used to describe the development of composite systems and methods for composite system development. We have concentrated on using the framework to handle the "multiple perspectives problem". Viewpoints provide a modular approach which reflects the structure of the "real world" systems development process.

In attempting to compare VOSE with other work a problem arises. The framework cuts across many existing approaches to support for system development. There is a substantial intersection with process modelling (e.g. Kaiser 1988), with view integration in software development environments (e.g. Meyers 1991), with generic or meta-CASE tools (e.g. Alderson 1991), with specification and design methods (e.g. Jackson 1990), with research on

multiple perspectives (e.g. Robinson 1989) and with multi-paradigm development (e.g. Zave 1989). There are also relevant contributions in method modelling (e.g. Potts 1989), formal specification of requirements (e.g. Hagelstein 1988) and distributed systems development (Kramer 1990). Short of reviewing all these areas it is difficult to provide a detailed comparative analysis of the approach. The key points distinguishing our approach are summarised below.

The framework is substantially different from many existing approaches to systems development support. In these, multiple “views” are integrated by the use of a common data model, usually supported by a common data base or repository. Extensions to this scheme are then supported by the provision of mappings to/from the common data model. These approaches are difficult to extend and, in implementation terms, difficult to support efficiently. By contrast viewpoints give a distributable, loosely coupled and extensible scheme for view integration.

Experience in building automated support for systems engineering (Finkelstein & Kramer 1991) has shown us th developers use many representations and move rapidly between them. They tend to tolerate and even exploit inconsistency between different representations as they build a specification. Consistency is only relevant at certain stages and should not therefore be enforced as a matter of course. The viewpoint framework, by interweaving consistency checks and work plan, provides explicit support for this.

Existing approaches to process modelling are, we believe, limited. Their primary limitation is that they often ignore (design) representation in favour of (design) process. We believe that at the fine-grain level at which guidance is most appropriate representation and process are closely intertwined.

Though the viewpoint framework is open and can be used to support any method, including for example transformation based approaches, it provides particular support for methods which deploy many simple, highly redundant, structured, visual and formal representations. We argue that such methods form the best basis for system development. In this we follow the trend away from “universal” languages and “global” reasoning schemes towards heterogeneity and local reasoning.

We are currently developing the √iewer and in particular extending support for check and guide actions. Experience gained in this work is providing us with a better idea of the requirements for notations in which to define the contents of slots. We hope to continue work on describing existing methods and testing the applicability of the framework on a variety of examples.

Acknowledgements

The authors would like to thank their colleagues and students who have contributed considerably to the ideas expressed in this paper. In particular thanks to Jeff Magee (Imperial College), Jim Huang (City University) and Peter Graubmann (Siemens Research Laboratories, Munich). We would like to acknowledge support from the SERC (SEED project) and the European Community (REX project)

References

Alderson, A. (1991); Meta-CASE Technology; Proc. European Symposium on Software Development Environments and CASE Technology; LNCS 509, pp 81-91, Springer Verlag.

Finkelstein, A. & Kramer, J. (1991); TARA: Tool Assisted Requirements Analysis; Loucopoulos P. & Zicari R (Eds.); Conceptual Modelling, Databases & CASE: an integrated view of information systems development; Addison-Wesley.

Hagelstein J. (1988); A Declarative Approach to Information Systems Requirements; Knowledge Based Systems; 1, 4, pp 211-220.

Jackson, M.A. (1983); System Development; Prentice Hall.

Jackson, M.A. (1990); Some Complexities in Computer-Based Systems and Their Implications for System Development; Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90); pp 344-351, IEEE CS Press.

Kaiser G., Feiler P. & Popovich S. (1988); Intelligent Assistance for Software Development and Maintenance; IEEE Software; V5 N3, pp 40-49.

Kramer, J. (1990); Configuration Programming - A Framework for the Development of Distributable Systems; Proc. IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90); pp 374-384, IEEE CS Press.

Meyers, S. (1991); Difficulties in Integrating Multiview Development Systems; IEEE Software, 8, 1, pp 49-57.

Potts, C. (1989); A Generic Model for Representing Design Methods; Proc. 11th International Conference on Software Engineering; pp 217-226, IEEE CS Press.

Robinson, W. (1989); Integrating Multiple Specifications Using Domain Goals; Proc 5th International Workshop on Software Specification & Design; pp 219-226, IEEE CS Press.

Zave, P. (1989); A Compositional Approach to Multi-Paradigm Programming; IEEE Software, 6, 5, pp 15-25.