ViewPoint Oriented Software Development

Anthony Finkelstein, Jeff Kramer, Michael Goedicke

Imperial College of Science, Technology & Medicine
(University of London)

Abstract

In this paper we propose a new approach to software development which explicitly avoids the use of a single representation scheme or common schema. Instead, multiple ViewPoints are utilised to partition the domain information, the development method and the formal representations used to express software specifications. System specifications and methods are then described as configurations of related ViewPoints. This partitioning of knowledge facilitates distributed development, the use of multiple representation schemes and scalability. Furthermore, the approach is general, covering all phases of the software process from requirements to evolution. This paper motivates and systematically characterises the concept of a "ViewPoint", illustrating the concepts using a simplified example.

Proc. of Third Int. Workshop on Software Engineering and its Applications, Toulouse, December 1990.

1 Motivation

Software development is a complex combination of activities. It requires a knowledge of the application domain, combined with expertise in the software development process. This software process demands knowledge of appropriate software development methods, specification techniques and languages. The key to managing these activities and the associated forms of knowledge is to structure and contain them so as to provide a partitioned, distributable organisation for the software development process, and a partitioned, distributable structure for the software specification. We believe that a common partitioning and structuring for these activities and knowledge forms is both possible and desirable.

In the following introductory sections we justify and elaborate on our belief that the software process and software structure should be combined in a single framework which supports multiple representation schemes and alternative method steps.

1.1 Combining Software Process and Software Structure

Developing software-in-the-large involves many participants, with experts in various aspects of software development and in various aspects of the application area. In addition, each participant may have different roles, responsibilities and concerns which may change and shift as the software develops and evolves. Participants have knowledge which they want to bring to bear on the development of the specifications. This knowledge will generally complement that of the other participants but may also overlap, interlock and conflict. We believe that any attempt to treat this process using techniques based on a single representation scheme, common schema or global reasoning are doomed to failure. It is essential that the distributed nature of the knowledge and participants be recognised and explicitly incorporated in the development process.

This presents us with some closely related problems. With all these participants how can we guide and organise the process of software development? How do we assign and maintain responsibilities? How can we allow each participant to see only that aspect or part of the "specification world" which is relevant to that participant's interests and responsibilities?

Following from this, how can we ensure that, if each participant uses a "bespoke" representation for eliciting, presenting and determining properties of relevant parts of the specification world, potential inconsistencies and conflicts between different participants are noted and resolved?

These problems are commonly treated separately - the first in so-called software process modelling languages, the second in specification language structuring schemes. We propose the use of ViewPoints as *both* an organising and a structuring principle in software development.

In outline, a **ViewPoint** (we use the distinctive capitals to denote our interpretation) captures a particular role and responsibility performed by a participant at a particular stage of the development process. The ViewPoint must encapsulate only that aspect of the application domain relevant to the particular role, and utilise a single appropriate scheme (or *style*) to represent that knowledge. Viewpoints are thus a combination of both a portion of the development process (*workplan*) and a portion of the software structure.

1.2 A Structural Framework for Software Development

A well known difficulty, which arises with all approaches to structuring in software development, is that of "structural transformation". What appears an appropriate structure for carrying out requirements analysis is not suitable for design. What appears an appropriate structure for carrying out design is not suitable for construction and reuse and so on. We argue that ViewPoints provide a generic and consistent structuring approach which accommodates all aspects of software development. In particular ViewPoints allow us to support the activities of requirements elicitation and formalisation at the up-stream end of software development and system evolution at the bottom end. These activities are generally ignored in conventional approaches to software development.

In this respect ViewPoints provide the means to structure and relate activities and representation schemes which are on the one hand solely directed towards a particular area (e.g requirements engineering) and on the other hand are directed towards the exchange of information and knowledge between these areas. This structure will be reflected in *single* ViewPoints and *configurations* of ViewPoints respectively.

1.3 Using Multiple Representations

Much effort has been devoted to developing ever richer and more sophisticated formal representation schemes. On the surface this appears to be a worthwhile enterprise - if a representation scheme is made more expressive the task of elicitation and specification should, in theory, become easier. This has however not proved to be the case:

the learning overhead in the use of these schemes is significant;

the development of such schemes is extremely difficult, in particular developing sound and adequate verification or proof schemes;

such schemes are often very different from the conventional (and reasonably well understood schemes) used in software engineering practice and consequently pose difficulties for technology transfer;

the richer the representation scheme the easier it is to write baroque and unreadable descriptions;

although an expressive representation scheme may theoretically permit validation of complex properties of a description(for example, generation of consequences using formal reasoning), in practice validation by inspection or automated reasoning is usually more difficult;

In contrast to the 'universal' language approach ViewPoints provide the framework for representing the information in a collection of different but related representation styles. This supports two important aspects. One is that it is necessary to structure the (specification-) information of systems in a modular way which is also reflected on in the previous section. The other aspect is that representation schemes can be chosen which best suit the problem or sub-problem at hand.

1.4 Building Methods Systematically

It may be observed that there are close parallells between our aims and what practitioners have sought to achieve in methods. Methods are, in the strict sense of the term, the collection and packaging of software development knowledge. Unfortunately methods have commonly been overlooked in current computer science in favour of specification techniques or novel development paradigms.

Methods attempt to combine software process with software structure by breaking down a "work plan" into steps and stages and associating these with elements in a (generally functional) decomposition. Methods aim at providing systematic coverage of the software development activities. Methods provide organised collections of simple representation schemes which are closely related and provide guidance, integrated with a work plan, for moving between these schemes.

This close relation between the representation schemes suggests that structuring ViewPoints can be used as a means of presenting a method and managing method-derived information. (in a sense modules 'manage' their data and a ViewPoint manages its local knowledge)

1.5 Motivation Summary

This section has presented our motivation for ViewPoints in four parts:

unifying models of software process and models of software structure;

developing an overarching structural framework for software development;

supporting the use of multiple representation schemes;

providing a systematic basis for constructing and presenting methods.

Pragmatically, the principle of a ViewPoint as an encapsulation of role and knowledge, using an appropriate representation scheme, is motivated by the need as far as possible to avoid any single governing representation, schema or reasoning. Viewpoints are a means of supporting scalability by partitioning domain knowledge, of providing for distributed development by partitioning the development process, and of avoiding complex representation schemes by specifying relations between multiple simpler schemes. Furthermore we believe that the provision of tool support for a particular ViewPoint is simplified by its constrained role.

The next section provides a clear characterisation for our notion of a ViewPoint, using examples to illustrate the concepts. The relationships between ViewPoints are then discussed, leading to the description of both software structure and the software process as configurations of ViewPoints. Finally, the implications of this approach are discussed.

2 Characterisation of a ViewPoint

This chapter provides a general characterisation of ViewPoints in isolation. Earlier research work which lead to the conception of ViewPoints is first provided, followed by a more refined characterisation. Since a ViewPoint is also a means to express a certain perspective on a problem or system one likes to have the possibility, for example, to see different parts of a problem or system from the same perspective. Thus a kind of 'Viewpoint Type' is required which can be used to create ViewPoints as instances of such a type. This concept of ViewPoint type is called *ViewPoint template*. The concepts of ViewPoint templates and instances are illustrated in an example.

2.1 Background Research

The concept of a ViewPoint is a synthesis of the concepts of "view" and "viewpoint" which were successfully exploited in other research projects. The TARA (Tool Assisted Requirements Analysis) research project [Kramer et al 87, 88a, 88b] provided us with considerable experience of and respect for the method CORE [Mullery 85, Stephens & Whitehead 85]. CORE is based round the notion of viewpoints which are its primary structuring vehicle. A CORE viewpoint is "something that does things" in the domain under consideration, akin to an agent or role. It also takes into account the way in which authority for making decisions about the specification is distributed. Thus the CORE viewpoint can be seen to be the source for *domain* decomposition.

The notion of views as partial specifications and as the principal basis for incremental construction of specifications has been fully developed in the PEACOCK [Goedicke et al 89a, Goedicke 89b] and PRISMA [Niskier et al 89a, 89b] projects. These projects have convinced us of the importance of selecting the *representation* to suit the particular ViewPoint specification task, and of subsequently combining representations.

In the FOREST project [Cunningham et al 85, Finkelstein & Potts 87] we saw the need to find a better way of constructing methods for requirements formalisation. This lead us to think of the representations tied to each *method step* as, in database terms, providing a "view" on the specification information.

The concept of point of view on which the IC~DC work [Finkelstein & Fuks 89] is based has been carried over from the TARA work. The significant enhancement to the concept of viewpoint brought out by the IC~DC project is the idea of a point of view as a "software development participant", that is as an active, autonomous and loosely coupled agent - in the distributed artificial intelligence style. This has raised the possibility of interpreting ViewPoints as active agents. Other influences on the approach we have adopted are those of "selfish views" [Robinson 1989] and "contexts" in ERAE [Finkelstein & Hagelstein 1989].

The concept of a separate, explicit structural (*configuration*) description for the software architecture of a system has been fully investigated in the Conic environment for developing distributed systems [Kramer et al 89a, Magee et al 89, Kramer 90a]. It has been shown to be essential for all phases in the software development process, from system specification as a configuration of component specifications to evolution as changes to a system configuration. The notion of forming "configurations" of ViewPoints is suggested by the need to provide an explicit structure for describing ViewPoint relations, and the interesting correlation between the configuration of ViewPoints used in the software process and the resulting software structure [Kramer et al 90b].

2.2 ViewPoint Definition and Characterisation

This background research lead to the formulation of a ViewPoint as

> *A ViewPoint is a loosely coupled, locally managed object which encapsulates partial knowledge about the application domain, specified in a particular, suitable formal representation, and partial knowledge of the process of software development.*

A ViewPoint is a combination of the following parts to which we refer to as slots:

> *a* ***style****, the representation scheme in which the ViewPoint expresses what it can see*
> (examples of styles are data flow analysis, entity-relationship-attribute modelling, Petri nets, equational logic, and so on);
>
> *a* ***domain*** *defines which part of the "world" delineated in the style (given that the style defines a structured representation) can be seen by the ViewPoint*
> (for example, a lift-control system would include domains such as user, lift and controller);
>
> *a* ***specification****, the statements expressed in the ViewPoint's style describing particular domains;*
>
> *a* ***work plan****, how and in what circumstances the contents of the specification can be changed;*
>
> *a* ***work record****, an account of the current state of the development.*

As can be seen, the ViewPoint encapsulates knowledge in the form of various slots e.g. a *style* and a *specification*. The slots style and work plan represent general knowledge, in the sense that it can be applied to a wide range of problems. In contrast to this the knowledge encapsulated in the slots *domain*, *specification* and *work record* of a ViewPoint represent specific knowledge related to one particular problem. The *specification* is given in a single consistent style and describes an identified *domain* of the problem area. The *work record* describes the current state of the specification with respect to the development activities and concerns of the ViewPoint. This would include interaction between viewpoints to transfer information and perform activities such as consistency checks.

ViewPoints are organised in *configurations* which are collections of related ViewPoints. A *ViewPoint template* consists of a ViewPoint in which only the style and the work plan have been defined. A *method* in this setting is a set of ViewPoint templates and their relationships, together with actions governing their construction and consistency. In the following we explain shortly the concept of ViewPoint template followed by related examples.

2.3 ViewPoint Templates and Instances

A ViewPoint *template* elaborates only the style and work plan slots. These aspects are closely related as the work plan describes the basic actions which need to be performed in order to provide a specification in the given style. As such, they are general, and can be used to guide the specification of any specific, selected portion of the application domain. Such a specification is termed a ViewPoint *instance* since it refers to a specific instantiation of the template, and would include identification of the selected domain and elaboration of the specification and its state of development, given as the work record. ViewPoint instances are henceforth referred to simply as ViewPoints wherever such use is unambiguous.

A *system specification* is thus a set of (consistent) specifications given in selected ViewPoint instances describing those parts of the domain which are of interest. Should the information in one ViewPoint be disjoint from those in others? In general, ViewPoint styles, and hence specifications, will overlap. As described above in section 1.3, it is certainly advantageous to describe the same domain using different styles to specify different aspects of behaviour. Similarly domains will tend to overlap. Although such redundancy in the specifications enhances the potential for consistency checking, it is clear that the identification and selection of interacting rather than overlapping domains (as is done in CORE viewpoints) simplifies the relationships between ViewPoints. This simplification facilitates the practical process of checking system consistency and of reasoning about system behaviour.

Below we give some examples of possible ViewPoint templates and ViewPoints for a simple library application. In a later section we will discuss more fully the relationship between ViewPoints and illustrate this by a further development of the library example.

2.4 Examples of Possible ViewPoint Templates

We first develop two ViewPoint templates ST and DF which allow us to conduct state transition analysis and data flow analysis respectively.

In defining a ViewPoint template, the selected style and associated work plan are described. These descriptions should be formal in order to provide for concise and precise specifications and to facilitate formal reasoning. Furthermore, the intention is to enable software tools to be provided to support the work plan in the development of specifications in the selected style. The formalism used for describing the work plan should avoid dictating a specific ordering for the actions. It should rather provide a partial ordering such as can be specified using action pre- and post-conditions. In addition to specifying the actions necessary for giving a specification in the selected style, the work plan should include any required actions for checking specification consistency (both intra- and inter-ViewPoint) and criteria for checking completion of the specification.

State Transition Analysis

The ViewPoint template for the representation and development of a system (or part of a system) using state transition diagrams is given in Table 1. It is simplified by excluding such features as hierarchical decomposition. The style which gives us the language in which to capture states and transitions is outlined first. This is followed by an outline of the work plan which is described in terms of actions which may be applied to state transition diagrams and axioms describing the relations between those actions. The representation scheme for state transition analysis is presented in terms of annotated directed graphs. In the specifications which follow we will, for clarity of exposition, use the graphic representation.

Note that the notation used in the examples to describe the workplan is **not** an integral part of the concept; other notations could be used. The selection of improved notations for this purpose is the subject of further work. In our simplified “first stab” at a notation axioms are expressions of the form PÆ[a]Q. This should be read (roughly) as ‘if condition P holds, then after action a is performed condition Q holds’.

Data Flow Analysis

In a similar way to that described above, we can describe the ViewPoint template for data flow analysis DF (Table 2). This is based on a simple version of data flow diagrams (excluding refinement and data dictionaries). The properties of a system (or part of a system) are described as a collection of functions, stores and terminals which are connected by data flows.

The representation scheme for data flow analysis is presented in terms of annotated directed graphs. In the specifications which follow, the graphic representation is again used for clarity of exposition.

	States	State	Set of Nodes State of symbols (represented by circles) denoting a state
Style	Transitions	Trans	Set Trans of labelled Edges given by E,L,T
		Transition names T Edges E Labelling L	T is a set of symbols denoting Transition names with $T \cap State = \emptyset$ with $E \subset State \times State$ with $L : E \rightarrow T$
Work Plan	basic actions	add_state, remove_state, add_transition, remove_transition	
	work plan actions	identify_boundary_states, identify_internal_states, identify_transitions, check_consistency	
	heuristics	more_than_one_transition_per_state_pair more_than_max_states	
	axioms	empty_diagram $\rightarrow$ [identify_boundary_states] boundary_states_identified boundary_states_identified $\rightarrow$ [identify_internal_states] all_states_found true $\rightarrow$ [identify_transitions] all_transitions_found all_states_found $\wedge$ all_transitions_found $\rightarrow$ [check_consistency] got_nice_ST_diagram $\vee$ ST_inconsistencies	

Table 1: ViewPoint Template ST for State Transition Analysis

Style	Terminals	Term	Set of nodes Term of symbols (represented by square) denoting a terminal node
	Functions	Func	Set of nodes Func of symbols (represented by circles) denoting function nodes
	Stores	Store	Set of nodes Stores of symbols (represented by two parallel horizontal lines) denoting data stores Term,Func,Store pairwise disjoint and let the set N of graph nodes $N = Term \cup Func \cup Stores$
	Data flows	Data flow	Set Data flow of labelled Edges given by E,L,F Data flow names FF is a set of symbols denoting Data flow names with $F \cap N = \emptyset$ Edges E with $E \subset N \times N$ Labelling Lwith L : $E \rightarrow F$
Work Plan	basic actions	add_node(type), remove_node, add_data_flow, remove_data_flow	
	work plan actions	identify_terminal_nodes, identify_function_nodes, identify_data_store_nodes, identify_data_flows, check_consistency	
	heuristics	more_than_max_functions? ...	
	axioms	empty_diagram $\rightarrow$ [identify_terminal_nodes] terminal_nodes_identified empty_diagram $\rightarrow$ [identify_function_nodes] function_nodes_identified empty_diagram $\rightarrow$ [identify_data_store_nodes] data_store_nodes_identified true $\rightarrow$ [identify_data_flows] all_data_flows_found terminal_nodes_identified $\wedge$function_nodes_identified $\wedge$ data_store_nodes_identified $\wedge$ all_data_flows_found $\rightarrow$ [check_consistency] got_nice_DF_diagram $\vee$ DF_inconsistencies	

Table 2: ViewPoint Template DF for Data Flow Analysis

2.5 Examples of Possible ViewPoints for a Library Application

While ignoring (for the present) the relationship between ViewPoints, we now give examples of the use of our templates in a small Library application description.

The Library ViewPoints:

In a library there are many people that play a part: users, librarians, inventory clerks, purchasers, and so on. In our example we will look at two parts of the library: the library desk (effectively the librarian's perspective) and the library user (Figure 1). Users take a book from the shelves, present it at the desk and, depending on the status of the book, it will be either lent to the user or sent to the part of the desk where reserved books are kept. If lent, a user then reads the book and returns it to the desk where it will, depending on the state of the book, be either given to someone else to process as a returned book or kept in the special place for reserved books. Books are released from reservation when claimed by the reserving user.

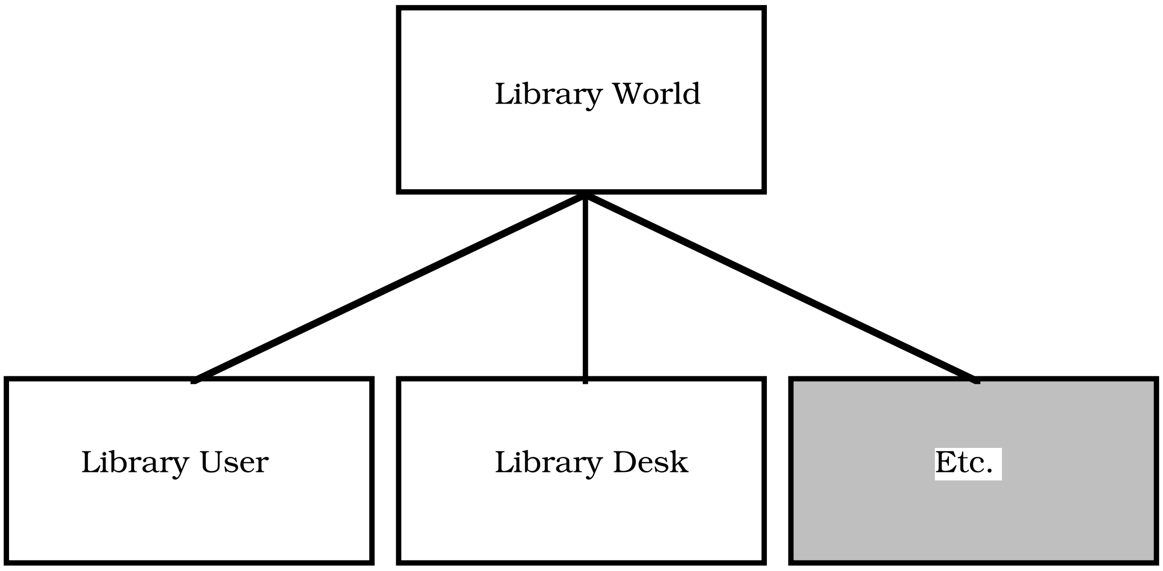


Figure 1: Simple library

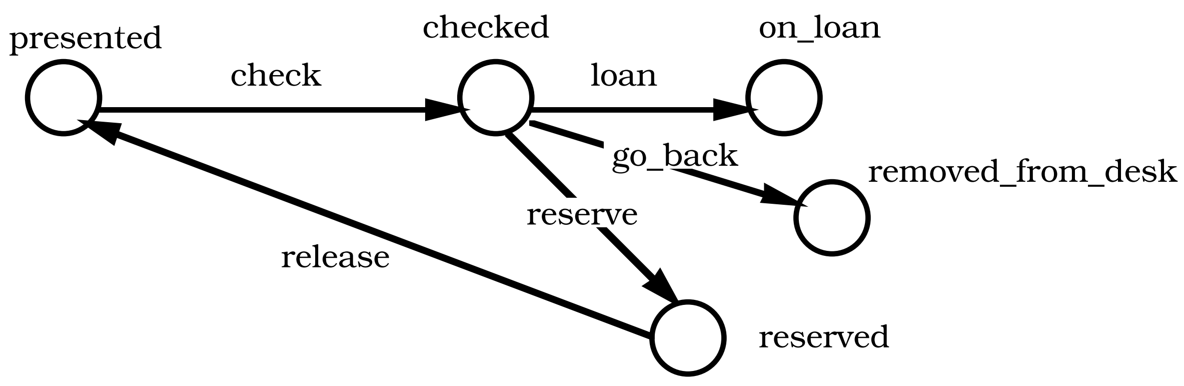
We now examine this fragmentary ViewPoint configuration for the library system. Our configuration will consist of three ViewPoints. Below we discuss the following ViewPoints in isolation first. *LDS (library desk, state transition analysis)*, *LDDF (library desk, data flow analysis)* and *US (library user, state transition analysis).* Thus we will develop two ViewPoints which refer to the same domain but are instances of different ViewPoint templates and one with a different domain.

Note that the ViewPoint configuration for the library system can itself be considered as a ViewPoint. It encapsulates a portion of the library world, described in a configuration of ViewPoints style and involves a workplan describing the decomposition and ViewPoint identification process.

Library desk, state transition analysis ViewPoint: *LDS*

The style and work plan of *LDS* is given by the ViewPoint template ST in table 1. We are interested in filling in the component slots of the ViewPoint not already covered by the template description. These are domain, specification and work record.

The domain of *LDS* is the library desk of the simple library. The ViewPoint cannot see states such as on_order or finished which are relevant only to the purchase department and library user respectively. The domain defines the boundaries of the knowledge encapsulated by the ViewPoint. The specification of *LDS* is shown below in Figure 2.



domain: Library_desk

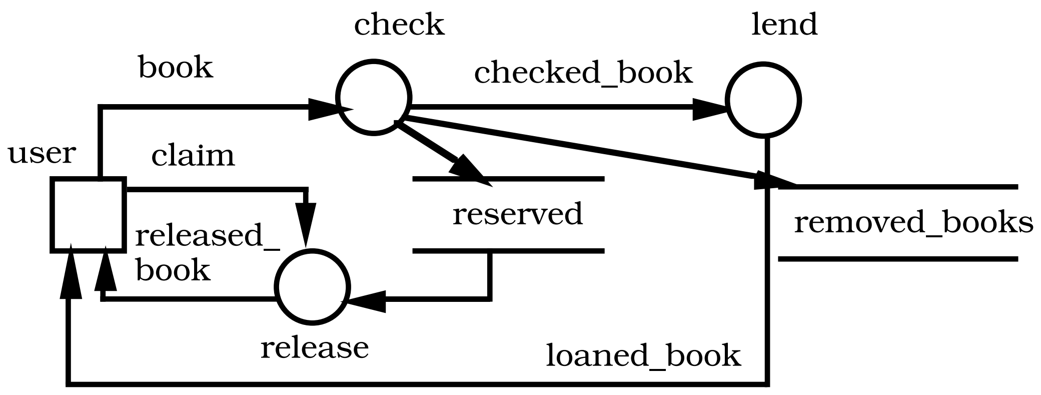
sees States: presented, on_loan, checked, removed_from_desk, reserved

Figure 2: State transition analysis specification of library desk domain

The actions which can be performed are given in the ViewPoint template description (add_state, add_transition were performed a number of times to arrive at this specification). The various occurrences of these actions are recorded in the work record. One of the main objectives is to provide a repository for capturing the design decisions taken during the specification process.

Library desk, data flow analysis ViewPoint: *LDDF*

The style and work plan are provided by the ViewPoint template DF in table 2. The specification of *LDDF* is shown in Figure 3.



domain: Library_desk

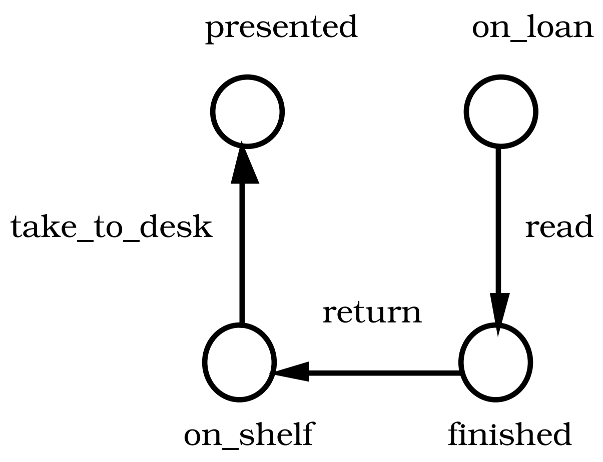
sees Functions: release, check, lend

sees Stores: reserved, removed_books

Figure 36: Data flow analysis specification of the library desk domain

Library user, state transition analysis ViewPoint:*US*

The style of *US* is the state transition analysis scheme given by ST in table 1. The domain of *US* is the library user (from whom the internal workings of the library desk are hidden). The specification of *US* is shown below in Figure 4. The work plan and work record are similar to that described for *LDS*..



domain: Library User

sees states: presented, on_loan, finished, on_shelf

Figure 4: State transition analysis specification of library user domain.

3. The Relationships Between ViewPoints

We will now discuss the nature of the relationships between different ViewPoints and between ViewPoint templates. At a first glance the number of different kinds of relationships might look unbounded. Indeed the theoretically possible combination between a number of ViewPoints is not small. Instead of exploring the nature of all mutual relations we would like to restrict ourselves to a small number of sorts of relationships.

In practice we believe that these relationships can be kept to a manageable number. ViewPoints are not selected arbitrarily: the domains obviously interact and are closely related, and the representation styles can and should be selected so as to express different aspects yet permit reasonable mappings between them. Our experience in TARA (with CORE), Peacock and Prisma confirm this point. This is further discussed below when discussing *methods*.

For example, at the requirements elicitation stage it is useful to use the ViewPoints to reflect the perspectives of different participants in the process of elicitation. Thus a ViewPoint is created for each relevant participant. This could lead to assigning a separate *domain* to each participant. Each ViewPoint is an instance of a ViewPoint template. Thus we have to describe the relations between ViewPoints of the same domain, but different ViewPoint template, and ViewPoints of the same ViewPoint template but describing different domains.

Using our library example this could be depicted as shown in figure 5. In this discussion we limit ourselves to the relations marked with solid lines.

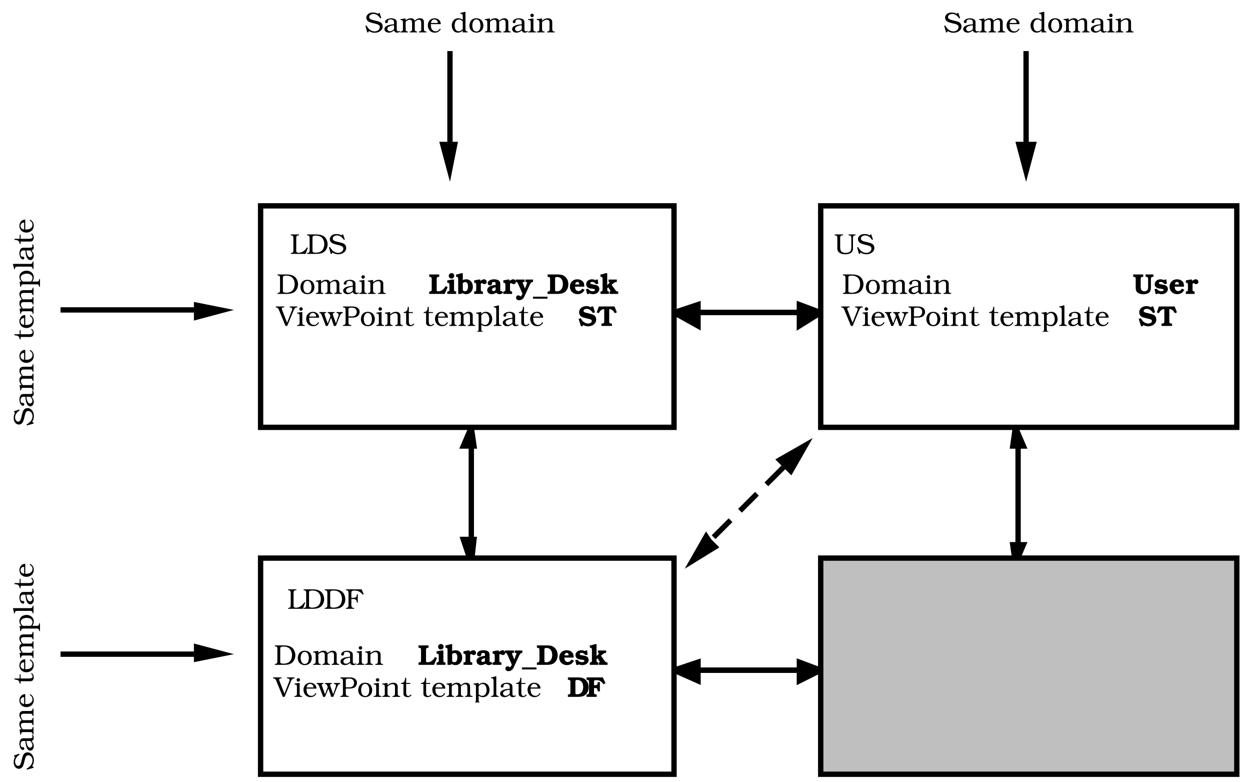


Figure 5: Library example, relations between ViewPoints

The relationships between ViewPoints describing the same domains but using different templates should be specified in the method as mappings and consistency checks between the representations. For example, a mapping from the state transition formalism to the data flow could map states in the former to data flows or data stores in the latter. Mappings of terms (eg. aliases) need to be specified for the domain by the ViewPoint instance. Such mappings have been shown to be useful and practical in Peacock and Prisma.

ViewPoints describing different domains but using the same templates provide a partitioned view of the domain, with relatively easy mappings specifiable by the ViewPoint instance. Our experience is that these mappings are easier if the domains are interacting rather than overlapping (as in CORE). For example, a check could ensure that a flow produced by one ViewPoint is consumed by another. However, such domain identification may not always be practical in a system under distributed development. More experience is needed.

Further, the descriptions using different styles may not correspond to the domain boundaries selected for the same style. We believe that it may be easier to manage ViewPoints if such a restriction is imposed. However, no such restriction is embedded in the general ViewPoint approach, and we realise that it may not be practical in many circumstances. For instance, timing analysis ViewPoints may well cut across different functionally specified ViewPoints.

3.1 Methods as Configurations of Viewpoint Templates

A *method* is a configuration of a selected set of ViewPoint templates which together describe the styles and work plans used in the method. The mappings and checks between templates should also be specified.The dynamics of the method are described by permitting one ViewPoint to create (or spawn) another as the method unfolds. Information in a "parent" ViewPoint which is relevant to "child" ViewPoint can be transferred using the mappings. A method is thus a dynamically evolving configuration of viewpoint templates.

The partitioning of the application domain provides another dimension to the method. For each domain the method may need to create a configuration of ViewPoints as the method unfolds.

3.2 An example: The Method NYCE

We now develop the NYCE (Not Yet Completed Example) method which consists of the ViewPoint templates ST and DF described above. To do this we need to consider the relations between ViewPoints of the same domain but based on a different template and the relations between ViewPoints based on the same template but different domain (we need, for example, to define the relations between two state transition analyses each representing a different part of the system). Figures 5 illustrates the relations which we will need to define in our example.

It should be noted that the art of developing a method is not to make all representations equivalent. We are not in the business of simply expressing the same properties of a system in another style but rather providing a combination of ViewPoint templates that give "beneficial" complementarity.

We use a ViewPoint template to describe the overall method. This may at first seem confusing but it allows us to give a uniform and systematic presentation.

Tables 3 & 4 show the style and work plan slot respectively for the method NYCE. We refer to the various components (functions, stores, and so on) within a ViewPoint template (such as that for data flow analysis) by the notation <ViewPointTemplateName>.<componentname>. Multiple ViewPoints based on the same ViewPoint template are distinguished by primes/dashes(DF', DF"). By writing DF".Data flow we refer to the symbols that denote data flow names of the ViewPoint DF".

Style	Trigger	Trigger $\subseteq$ ST.Trans $\times$ DF.Func	A transition in a state transition analysis may correspond to a function in a data flow analysis. In this case the occurrence of such a transition is seen as a trigger for the corresponding function
	Data_ Condition	Data_Condition $\subseteq$ ST.State $\times$ Store $\cup$ DF.Data flow.F	A state in a state transition analysis may correspond to a Data flow or a Store in a Data flow analysis.
	Same_State	Same_State $\subseteq$ ST.State $\times$ ST'.State	Two state transition analyses may correspond to each other in terms of their respective state. This defines the states which are in common between two different state transition analyses

Table 3: Style slot for method NYCE

The first two of these relations are shown in Figures 6 and 7. States are represented as circles and transitions as arrows while in data flow diagrams functions are represented as circles and data flows as arrows respectively.

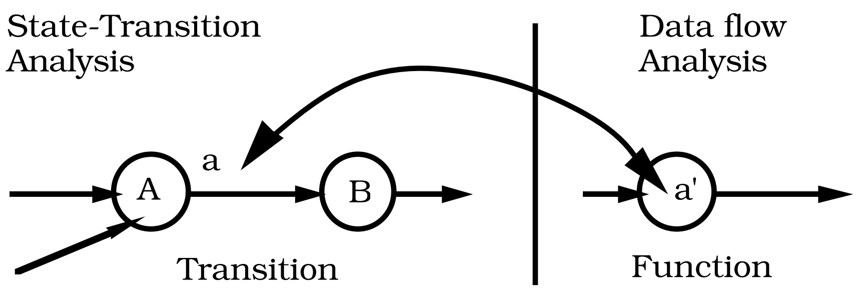


Figure 6: A possible relation between transitions and functions

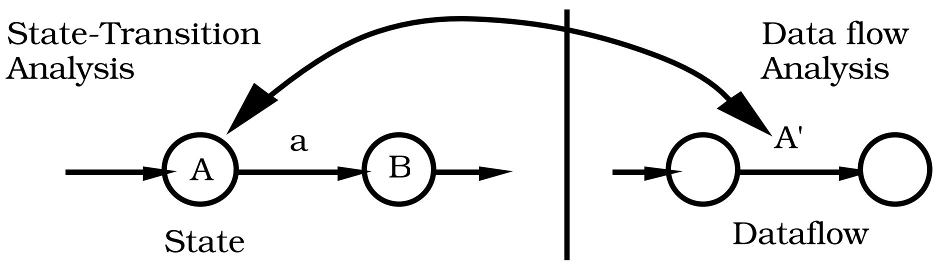


Figure 73: A possible relation between states and data flows

As we emphasised before it is not necessarily the case that for every data flow there is a corresponding state or vice versa. If it were so, the two representation schemes would degenerate to a common representation.

Work Plan	basic actions	create_DF_ViewPoint(*DOMAIN,NAME*), create_ST_ViewPoint(*DOMAIN,NAME*),
	work plan actions	list_relevant_domains → *Set of Domain*, check_consistency(*NAME*), relate_ViewPoints_same_domain(*DOMAIN,Set_of_NAME*), relate_ST_ViewPoints(*Set_of_NAME*), relate_DF_ViewPoints(*Set_of_NAME*),
	heuristics	any_ViewPoint_too_complex, possible_conflicts_made_explicit, resolve_conflict,...
	informal	1. list_all_relevant_domains 2. for all domains perform in parallel create_ST_ViewPoint and create_DF_ViewPoint 3. check_consistency locally for each ViewPoint created in step 2 4. relate_ViewPoints pairwise by either relate_ViewPoints_same_domain, relate_ST_ViewPoints or relate_DF_ViewPoints respectively 5. if any conflict found then resolve_conflict and start with step 3 again 6. if any_ViewPoint_too_complex then try to split the ViewPoint and start with Step 3 again

Table 4: Work plan for method NYCE

3.3 Configurations of ViewPoints: Completion of the Library Example

After having introduced the various notions included in the ViewPoint concept we are now in the position to complete our example. The method NYCE which is based on the two templates ST and DF is used to establish and relate the three ViewPoints developed in the previous text. We describe how the ViewPoints LDS, LDDF and US are related to each other by the means the method NYCE provides.

In Tables 3 & 4 which defines the method NYCE we have set out the rules governing the relation between state transition analysis and data flow analysis. Thus Trigger and Data_Condition relate ViewPoints of the same domain. In the case of ViewPoints of different domains based on the same ViewPoint template we define the relation called Same_State to express the overlap of two state transition analyses[1] . To capture the relation between

[1]This is, of course highly simplified as several states in one state transition analysis may correspond to one state in another

the actual ViewPoints the relationship between the ViewPoints *LDS* and *LDDF* is defined by giving an instance for the relations of type Trigger and Data_Condition respectively since these ViewPoints are of the same domain but different templates as for example Table 5. This states that the transitions check, loan and release correspond to the functions check, lend and release of the data flow analysis.

LDS-LDDF.Trigger ($\subseteq$ LDS.Trans $\times$ LDDF.Func)	
LDS.Trans	LDDF.Func
check	check
loan	lend
release	release

Table 5: Trigger relationship for the ViewPoints LDS & LDDF

The other correspondence relation describes the data in more detail. An instance of the relation Data_Condition is given below in Table 6.

LDS-LDDF.Data_Condition ($\subseteq$ LDS.State $\times$ Store $\cup$ LDDF.Data flow)	
LDS.State	LDDF.Store $\cup$ LDDF.Data flow
presented	book
checked	checked_book
reserved	reserved
removed_from_desk	released_book
on_loan	loaned_book

Table 6: Data_Condition relationship for the ViewPoints LDS & LDDF

These relations (Table 5 & 6) define the close relationship of the two library desk ViewPoints and link the state transition perspective with a functional perspective on this domain.

Clearly we must also describe the relationship between the ViewPoints *LDS* and *US*. That is we need to say how different parts of the system -the library desk and the library user - overlap. We must define their respective roles within the library world. We can do this by giving an instance of the relation **Same_State** as in Table 7. This relation establishes that the overlapping states are presented and on_loan (the user does not see the internal workings of the library and vice-versa).

LDS-US.Same_State (⊆ LDS.State × US.State)	
LDS.State	US.State
presented	presented
on_loan	on_loan

Table 7: Same_State relationship for ViewPoints LDS & US

4 Conclusions

In this paper we have proposed a new approach to software development in which multiple ViewPoints are utilised to partition the domain information, the development method and the formal representations used to express software specifications. System specifications and methods are described as configurations of related ViewPoints.

We believe that ViewPoints provide a basis for unifying models of the software process and models of software structure, as exemplified in the "configuration programming" [Kramer 90a]. The partitioning of knowledge exemplified in the ViewPoints approach facilitates distributed development, the use of multiple representation schemes and scalability. Furthermore, the approach is general, covering all phases of the software process from requirements to evolution.

An additional benefit which seems to follow from the identification and encapsulation of style (representation) and workplan (specification method) in a single ViewPoint Template is the opportunity for tool support. Individual support could be designed for each template in a particular method, thereby simplifying the complexity of the tool in much the same way as one expects to simplify the steps and expression of that particular ViewPoint specification. We can then envisage method tool support as comprising a configuration of template support tools, configured to suit the particular method adopted.

The work on ViewPoints which this paper reports is in its early stages and requires considerable further work. A major objective is to complement our intuitive use of ViewPoints with a comprehensive formal description. We are investigating the use of M[A]L [Khosla & Maibaum 89] as a suitable base for such a description.

We believe that ViewPoints provide a systematic basis for constructing and presenting methods. ViewPoints would be particularly useful in the description of mixed approaches such as those described as "multiparadigm programming" [Zave 89]. The ViewPoint approach is also strongly related to Jackson's recent work on views and implementations [Jackson 90] in which he describes "complexity in terms of separation and composition of concerns", and focuses on the problems of coping with the relationships between concerns (cf. ViewPoint relationships).

Our short term goal includes developing descriptions, in the ViewPoint style, of a repertoire of standard software development methods such as SSADM and JSD. This would act as a means of refining the ViewPoint concept and of illustrating the utility of the approach. In the longer term we intend to develop a ViewPoint based method for developing reconfigurable and extensible distributed systems [Kramer et al 90b].

Acknowledgements

The authors would like to thank their colleagues and students; in particular Celso Niskier, Tom Maibaum, Jeff Magee and Hugo Fuks, for the lively critical discussion which has contributed significantly to the work this paper reports. We also thank Michael Jackson for his comments and suggestions which have helped to clarify the paper. Michael Goedicke was a Visiting Research Fellow on leave from the University of Dortmund, FRG.

References

Cunningham J., Finkelstein A., Goldsack S., Maibaum T. & Potts C. (1985); "Formal Requirements Specification - The Forest Project"; Proc. 3rd International Workshop on Software Specification & Design; pp 186-191, IEEE CS Press.

Finkelstein A. & Fuks H. (1989); "Multi-Party Specification"; Proc 5th International Workshop on Software Specification & Design; pp 185-196, IEEE CS Press [Also ACM Software Engineering Notes May 1989].

Finkelstein, A. & Hagelstein, J. (1989); "Formal Frameworks for Understanding Information System Requirements Engineering: a research agenda"; Nijssen, S. & Twine, S.(Eds) IFIP CRIS Review Workshop; North-Holland.

Finkelstein, A. & Potts, C. (1987); Building Formal Specifications Using "Structured Common Sense"; Proc. 4th International Workshop on Software Specification & Design; IEEE CS Press.

Goedicke M., Ditt W., Schippers H. (1989a); "The ∏-Language Reference Manual", Research Report No 295 1989, Department of Computer Science, University of Dortmund.

Goedicke,M.(1989b); "Paradigms of Modular Software Development" (to appear) Mitchell R.J. (Ed); Managing Complexity in Software Engineering; Peter Peregrinus, Stevenage, England

Jackson M.A., "Some Complexities in Computer-Based Systems and their implications for System Development", Proc. of IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90), Tel-Aviv, Israel, May 1990, 344-351.

Khosla S., Maibaum T. (1989); “Time, Behaviour and Function”; (In) Barringer H., Proc. Colloquium on Temporal Logic & Specifications (Handbook of Temporal Logic); LNCS Springer Verlag.

Kramer J., Finkelstein A., Ng K., Potts C. & Whitehead K. (1987);"Tool Assisted Requirements Analysis: TARA final report”; Imperial College, Dept. of Computing, Technical Report 87/18.

Kramer, J., Ng, K., Potts, C., and Whitehead, K. (1988a), “Tool support for Requirements Analysis”, IEE Software Engineering Journal 3(3), (1988), 86-96.

Kramer, J., and Ng, K. (1988b), "Animation of Requirements Specifications", Software - Practice and Experience, 18(8), (1988), 749-774.

Kramer, J., Magee J., and Sloman M. (1989a), "Configuration Support for System Description, Construction and Evolution", Proc. of IEEE 5th Int. Workshop on Software Specification and Design, Pittsburgh, May 1989.

Kramer, J., Magee, J., and Ng, K. (1989b),"Graphical Configuration Programming", IEEE Computer, 22(10), (1989), 53-65.

Kramer, J. (1990a), “Configuration Programming - A Framework for the Development of Distributable Systems”, Proc. of IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90), Tel-Aviv, Israel, May 1990, 374-384.

Kramer,J., Magee, J., and Finkelstein, A. (1990b), "A Constructive Approach to the Design of Distributed Systems", Proc. 10th IEEE Int. Conf on Distributed Computing Systems, Paris, June 1990.

Magee J., Kramer J. and Sloman M. (89), "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.

Mullery, G. (1985); “Acquisition - Environment”; (In) Paul, M. & Siegert, H. "Distributed Systems: Methods and Tools for Specification"; Springer Verlag LNCS 190.

Niskier C. & Maibaum T. (1989a); “Acquisition, Classification and Formalisation of Software Specification Heuristics”; Proc. 3rd European Knowledge Acquisition Workshop.

Niskier C., Maibaum T. & Schwabe D. (1989b); “A Look Through PRISMA: towards knowledge-based environments for software specification”; Proc 5th International Workshop on Software Specification & Design; pp 128-136, IEEE CS Press [Also ACM Software Engineering Notes May 1989].

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

Stephens, M. & Whitehead, K. (1985); "The Analyst — A Workstation for Analysis and Design"; Proc 8th ICSE; IEEE CS Press.

Zave P. (1989), "A Compositional Approach to Multi-Paradigm Programming", IEEE Software, September 1989.