A Configurable Framework for Method and Tool Integration

Jeff Kramer and Anthony Finkelstein

Department of Computing,

Imperial College of Science, Technology and Medicine,

180 Queen's Gate, London SW7 2BZ, UK.

jk@doc.ic.ac.uk , *acwf @doc.ic.ac.uk*

ABSTRACT

There is an urgent need to provide a sound generic framework for method and tool integration, where many differing notations are used, software development is distributed and management support for the software development process is provided. This paper argues that there is much to be learnt from proven practical techniques for software construction, particularly those that support distributed software integration, heterogeneity and software management. *Configuration Programming* is one such approach which advocates the use of a separate, declarative configuration language for the description of system structure. It has been used in the Conic Environment for the development of distributable software, and is being extended for the configuration of heterogeneous components programmed in different programming languages. A number of software tools exist for the development, construction and management of Conic systems. This paper shows how an analogous set of the principles, practice and tools from configuration programming can be combined with recent work on ViewPoints[1] to provide a configurable framework for method and tool integration.

Invited Paper: European Symposium on Software Development Environments and CASE Technology, Königswinter, Germany, June 1991.

[1] The work on ViewPoints [6] has been conducted in close collaboration with Michael Goedicke of the University of Dortmund, but the obsession with configurations presented in this paper is that of the authors.

1. INTRODUCTION

The process of software production involves many stages, from requirements elicitation and specification through to system construction and maintenance. A large number of methods can be used during this process, each covering different stages of the process. Each method generally consists of one or more representation schemes (notations) together with a set of recommended procedures and heuristics as to how to complete each representation and guidance on how to move to the next representation. These methods may overlap or be disjoint. It is left to the project team to try to convert information to a suitable form for the next stage, to ensure consistency and to try to bridge the gaps between methods and notations by some *ad hoc* means. In addition, even at a single stage in the process, it is necessary to represent different aspects of the application, not only in terms of a partitioning of the application domain but also to provide different views (such as functional, performance, fault tolerance, safety and others). There is no doubt that different companies and project personnel will also wish to tailor individual methods and the overall software process according to their experience and the particular application domain.

Thus there is a need for multiple methods and their notations, together with the ability to modify and integrate them. The vision is of a framework for method integration, which supports distributed development by teams of personnel and provides tool support. How can methods be "glued" together? How can information transfer and consistency checks between notations be supported? How can advice and tool support for the use of each notation and between notations be provided?

The ViewPoints Approach

This paper addresses these issues and proposes a configurable framework for method and tool integration. The approach is based on decomposition of the application domain, the notations and the method steps. The basic entities which are configured in this framework are ViewPoints [6]. Each ViewPoint (fig.1) describes a part of the application *domain* using a single *style* (representation notation or formalism). A *workplan* provides the rules and heuristics for use of the notation together with relations or mappings to other ViewPoints to express information transfer and consistency checks. The result is a ViewPoint *specification* of a particular part of the application in a particular notation, together with its *workrecord* which records the current status of the specification elaboration.

Style : definition of representation notation
Domain : selected part of the application
Specification in the style for the particular domain
Work Plan : style use and interaction rules and heuristics
Work Record : specification status and history

Figure 1. A ViewPoint

Loosely speaking, a method is described as a configuration of “interacting” ViewPoint types (templates). Method use involves ViewPoint instantiation to provide the specifications in each notation. Method integration is at the ViewPoint level, by configuration to form a software process. Each ViewPoint can be elaborated separately subject to interaction constraints. Tool support can be provided for each type of ViewPoint and for configuration and interaction.

Why is it that we believe that configuring ViewPoints provides a sound generic framework for method and tool integration? In this paper we argue that there is much to be learnt from proven practical techniques for software construction, particularly those that support distributed software integration and interaction, heterogeneity and software management.

Learning from Software Configuration

Practical experience in software engineering has taught us that complex systems can be built and managed provided we adhere to sound principles. Software modularity is essential to encapsulate functionality behind clearly defined interfaces through which components can interact with their environment. Descriptions of the constituent software components and their interconnection patterns provide a clear and concise level at which to specify and design systems, and can be used directly by construction tools to generate the system itself. This approach has been variously referred to as "programming-in-the-large" [3], component-based system building using module interconnection languages [9,20], and "configuration programming" [15, 18]. Furthermore, evolution of the system can be achieved by making extensions or changes to the system configuration by the addition or replacement of components [16].

Configuration programming [18] advocates the use of the interconnected-component model for software design and construction through to evolution. The description of system structure (configuration), as a set of components and their interconnections, is separated from the functional description of individual component behaviour. This general approach has been

successfully used in the Conic Environment for the development of distributable software. Graphical and textual software tools are provided to construct, manage and evolve software systems so that they correspond to their declarative configuration descriptions. Complex components can be composed as interconnected instances of simpler component types. The approach is considered "constructive" since it emphasises the satisfaction of system requirements by composition of components. Provided that components adhere to interface standards, the configuration framework can also be used to construct systems from heterogeneous components ie. components written in different languages. Although the configuration descriptions vary from one system to another, many of the basic component types pertaining to an application domain tend to be the same. System variation is directed mainly at the configuration level, with some variation being embedded in particular components. Thus the opportunities for the employment of reusable components within an application domain are excellent, with the configuration language providing the means to select and tailor their use to the particular task at hand (cf. program families from information hiding modules [28]).

The analogy to configurations of ViewPoints and their interactions is obvious. We see ViewPoints as the means for encapsulating each aspect and notation, and configurations as the means for gluing them together. The description of method and process structure as a configuration is separate from the description of a primitive ViewPoint. In order to provide for the description of methods at different levels of abstraction, complex ViewPoints can also be defined as compositions of simpler ones. We believe that the configuration level provides a convenient level of abstraction at which to view and manage methods and software projects. In addition, its declarative structural form makes it independent of the procedural aspects, thereby implicitly supporting distributed (concurrent) development except where explicitly constrained by dependencies.

In this paper we describe the principles which underlie the configuration programming approach and briefly illustrate their usefulness by examples of some of the features of the Conic environment. The vision of the configuration framework for method integration using ViewPoints is then presented, based on analogous principles.

2. CONFIGURATION PROGRAMMING

2.1 Basic Principles

The basic principles of the configuration programming approach can be summarised as follows:

1. *The configuration language used for structural description should be separate from the programming language used for basic component programming.*
   This separation of concerns facilitates the description, comprehension and manipulation,

both by man and machine, of the system in terms of its structure. This is achieved by abstracting away from the component programming concerns.The structural nature of the configuration specification makes it amenable to both textual and graphical description. System construction can be performed by translation of the structural configuration description by component creation and interconnection. Furthermore, the configuration language should be *declarative*, describing what the structure is, not how it is to be constructed. Declarative descriptions tend to be more concise and amenable to analysis, interpretation, and manipulation than their imperative equivalents.

2. *Components should be defined as context independent types with well-defined interfaces.*
   Context independence [12] means that the component makes no direct reference to any non-local entities, but can be integrated into any compatible context without redefining or recompiling it. We therefore require that components access only local data and use indirect naming (such as local ports) to refer to connected components. Definition as a type permits instantiation and reuse in different contexts. The component interface should describe the interaction points with other components and permits validation of interconnections at configuration time.

3. *Using the configuration language, complex components should be definable as a composition of instances of component types.*
   Hierarchies are a natural and convenient means for the support of subcomponent encapsulation and information hiding. Interconnected instances of more basic component types can be composed to form more complex components (ie. an *instance* hierarchy). These composite components should themselves be component types, available for use in further definitions. Such an approach also permits the definition and construction of recursive structures.

4. *Change should be expressed at the configuration level, as changes of the component instances and/or their interconnections.*
   This follows from the first principle. Given that it is beneficial to utilise a structural description to comprehend and manipulate the system, then change can also be beneficially expressed as structural change. Changes can be made to component instances, which are then of a new and different type.

2.2 An Exemplar of Configuration Programming: Conic Environment

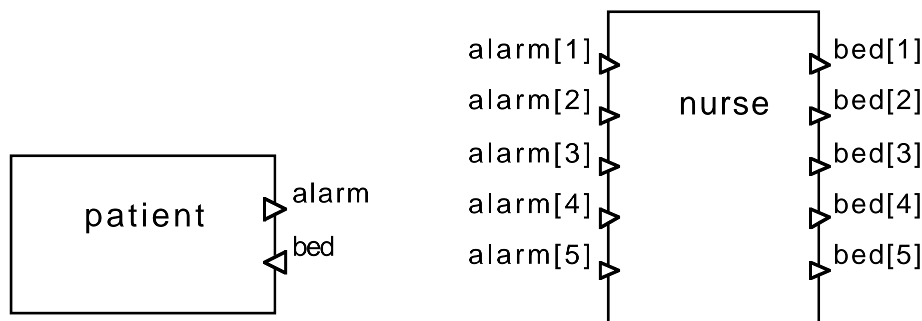
The Conic environment [12, 22], developed by the Distributed Computing Group at Imperial College, provides support for configuration programming for distributed and concurrent programs based on the above principles. The environment provides support for two languages,

one for programming individual components (processes) with explicitly defined interfaces, and one for the configuration of programs from groups of components. The Conic configuration language includes facilities for hierarchic definition of composite components, for parametrisation of components, for conditional configurations with evaluation of guards at component instantiation, and even for recursive definition of components [4]. In addition, the environment provides support for dynamic configuration using on-line management tools which permit dynamic creation, control and modification of application programs.

We now briefly illustrate some of the features of the Conic use of configuration programming for describing, constructing, monitoring and changing distributable systems. In order to provide a feel for the approach, we use a simple example: a patient monitoring system [24]. The intensive care ward in a hospital consists of a number of beds. Patients in each bed are continuously monitored for a number of factors, such as pulse, temperature and blood pressure. For each patient the current readings can be displayed both at the bedside and at the nurse unit. If any of the factor readings of a patient are outside of preset limits, then an alarm is sent to the central nurse station.

Component Types*: - Provision of context-independent components (principle 2).*

The patient monitoring system is constructed from the two context independent component types (referred to as *modules* in Conic) defined both graphically and textually below in Figure 2. The interface to a component is defined by typed exit- and entryports. Messages are sent out via **exitports** and received from **entryports**. The type definitions for messages and ports are imported from definition modules by the **use** clause




```
group module patient;

  use monmsg: bedtype, alarmstype;
  exitport
      alarm:alarmstype;
  entryport
      bed:signaltype reply bedtype;

      Periodically reads patient sensors.
```

```
group module nurse (maxbed:integer=5);

  use monmsg: bedtype, alarmstype;
  entryport
      alarm[1..maxbed]:alarmstype;
  exitport
      bed[1..maxbed]:signaltype reply bedtype;
```

Readings outside range cause alarm messages to be sent to *alarm*.
Request message received on *bed* returns current readings and ranges.
end.

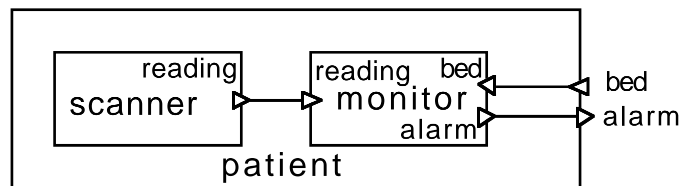
Displays alarms received from *alarm[]*

Requests particular patient data via *bed[]*
end.

Figure 2. The Patient and Nurse Component Types

Component Hierarchies: *- Composition of component instances (principle 3).*

In the above, we have described the main component types to be used to construct the patient monitoring system. In fact, each of the two component types used are themselves configurations of components. For example, the internal structure of the patient component is depicted in Figure 3. It is defined by instantiating an instance of each of a scanner and monitor component types and interconnecting their exit- and entryports. The links between exitports and entryports allow components to communicate by message passing. The Conic environment permits only ports of the same type to be connected.




```
group module patient;
    use monmsg: bedtype, alarmstype;
    use scanner, monitor;
    exitport    alarm:alarmstype;
    entryport bed:signaltype reply bedtype;
    create
        scanner;
        monitor;
    link
        scanner.reading to monitor.reading;
        bed to monitor.request
        monitor.alarm to alarm;
end.
```

Figure 3. Internal structure of Patient Module

A system in Conic is thus an hierarchic structure of component instances. The components at the bottom of the hierarchy are sequential tasks, implemented in a programming language. In Conic, the internal programming language is Pascal extended to support message passing for the distributed environment. Instances of these task modules execute concurrently.

Constructing Systems in Conic: - *Component instantiation and interconnection using a separate configuration language (principle 1).*

We can construct an initial patient monitoring system consisting of one nurse and one patient by instantiating one instance of each of the above component types and interconnecting their exit- and entryports. Again, the Conic environment permits only ports of the same type to be connected. The configuration description for this initial system is again shown both textually and graphically in Figure 4.

The system is created by submitting the configuration description to a configuration manager tool which downloads and interconnects component code. The configuration management tool and its supporting environment is described in [22]. In addition to instance creation and linking (interconnection), the configuration description can include component location (omitted here) and parameters. For example the nurse has a default parameter setting to the value 5 (Figure 2) however this could have been changed when the instance was specified, eg. **create** nurse: nurse(3).

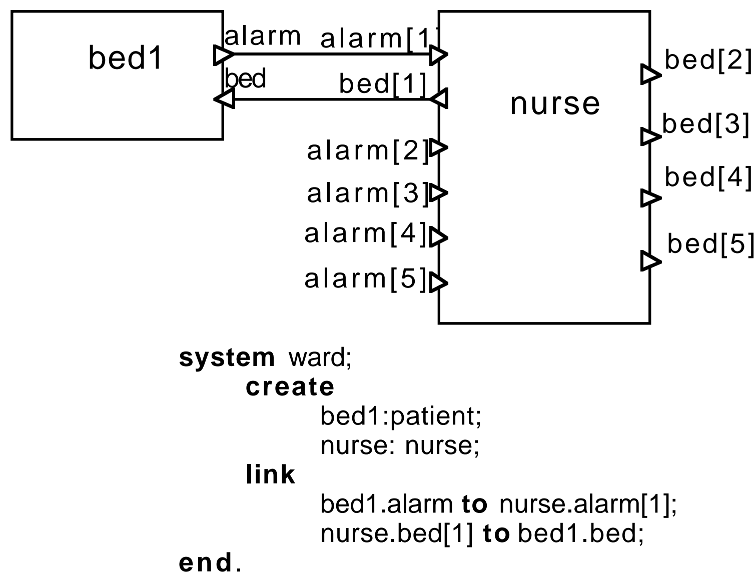


Figure 4. Initial Patient Monitoring System

REX, an ESPRIT II project, is extending the approach to permit configuration of heterogeneous components written in different programming languages.

Note that the use of a declarative configuration language enables the actual order in which configuration operations are performed to be left to the underlying support system. It can then exploit the inherent parallelism of the underlying architecture where appropriate. If the configuration statements were embedded in a procedural language, the current state of the system configuration would depend on the state of the configuration program (as in [19]). This would complicate the provision of support for dynamic configuration and the user/management view of the system, both of which are discussed below.

```
change ward;
      create
            bed2:patient;
      link
            bed2.alarm to nurse.alarm[2];
            nurse.bed[2] to bed2.bed;
end
```

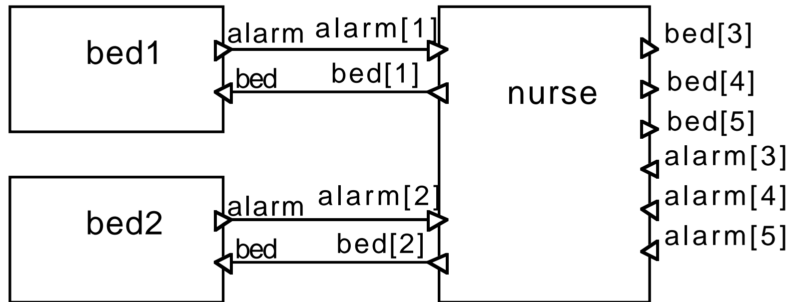
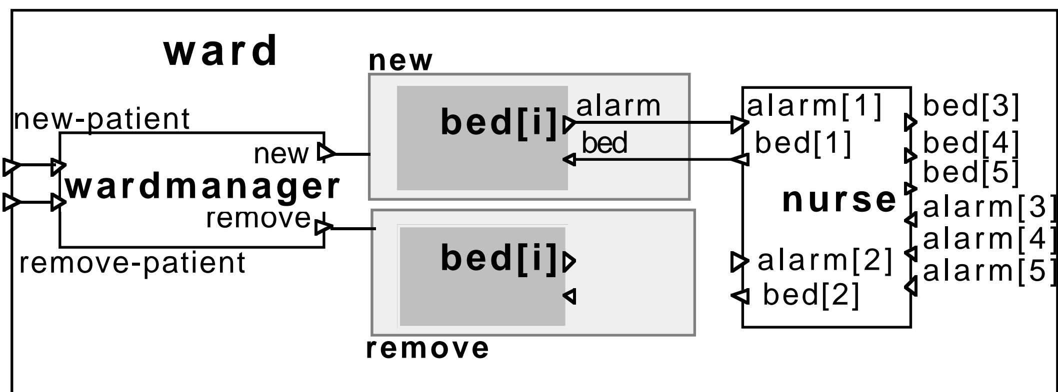


Figure 5 - Extended Patient Monitoring System

Dynamic Configuration for System Evolution:- *(principle 4).*

In addition to programming initial configurations, the Conic toolkit permits dynamic configuration: changes to running systems. For example, extending the above system to include an additional patient unit can be performed by submitting the configuration change of the system 'ward' (figure 5) to a configuration manager. The change can be thought of as an edit, in configuration terms, of the configuration specification *and* the system itself. It results in both a new specification and a correspondingly changed system. Thus the system itself can evolve rather than necessarily regenerating the system *ab initio*. Recent work on change management [16] has provided a sound basis for controlling change while preserving consistency and without disrupting the unaffected components.




```
component  ward (N:integer=5);

      entryport new-patient, remove-patient;
      use       patient, nurse, manager;
```

```
        create
                wardmanager:manager ;
                nurse: nurse(N) ;
        link
                wardmanager.new-patient         to new-patient;
                wardmanager. remove-patient to remove-patient;
                wardmanager.new       to new;
                wardmanager.remove to remove;

        change new (i:1..N);
                create bed[i]:patient;
                link  bed[i].alarm to nurse.alarm[i];
                link   nurse.bed[i]} to bed[i].bed;
        end;

        change remove (i:1..N);
                remove bed[i];
        end;

end.
```

Figure 6. Patient Monitoring System with Programmed Changes

The REX project is again extending this work to permit the definition of programmed reconfiguration changes to a composite component at the same level (and scope) as the definition of the component. What changes are permissible are defined in the configuration language; components determine when the changes should occur by invoking them. For instance, figure 6 shows a possible description of a ward component in which a wardmanager can invoke changes to create or remove patients, either under some internally specified condition or as a result of a request initiated from outside the ward (ie. by a request to new-patient). There is a general need to serialise the changes to a component to prevent interference between concurrent changes.

Tool Support: Graphical Configuration Monitoring and Management

In addition to language compilers, runtime environments and communications support, the Conic environment has provided software tools for monitoring system structure and component status, for dynamic configuration and even for “spying” on the message contents of a particular connection. We concentrate here on a graphics tool, ConicDraw, which can be used to display and manage system configurations. As depicted in figure 7, ConicDraw maintains a graphic representation of executing Conic systems in terms of the component instances which exist in the system, their interconnections and their execution state. It gathers this information directly from the executing system by communicating with a configuration manager.

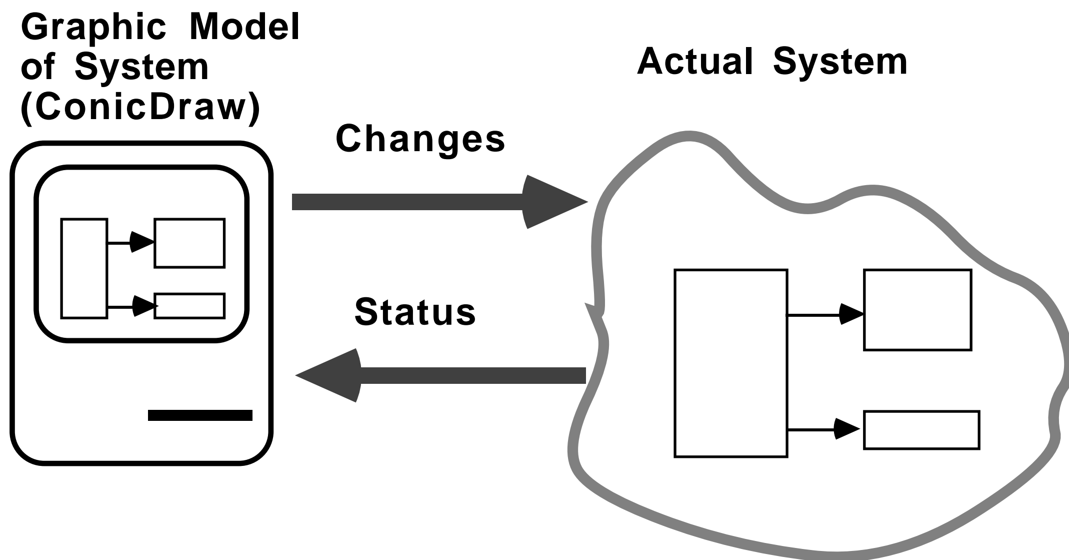


Figure 7. Interaction between ConicDraw and an Operational System.

Changes to the system are reported to ConicDraw by configuration management to enable it to maintain an uptodate view of the system. In addition, ConicDraw can itself instigate changes to the system as a result of edits to the graphic representation.

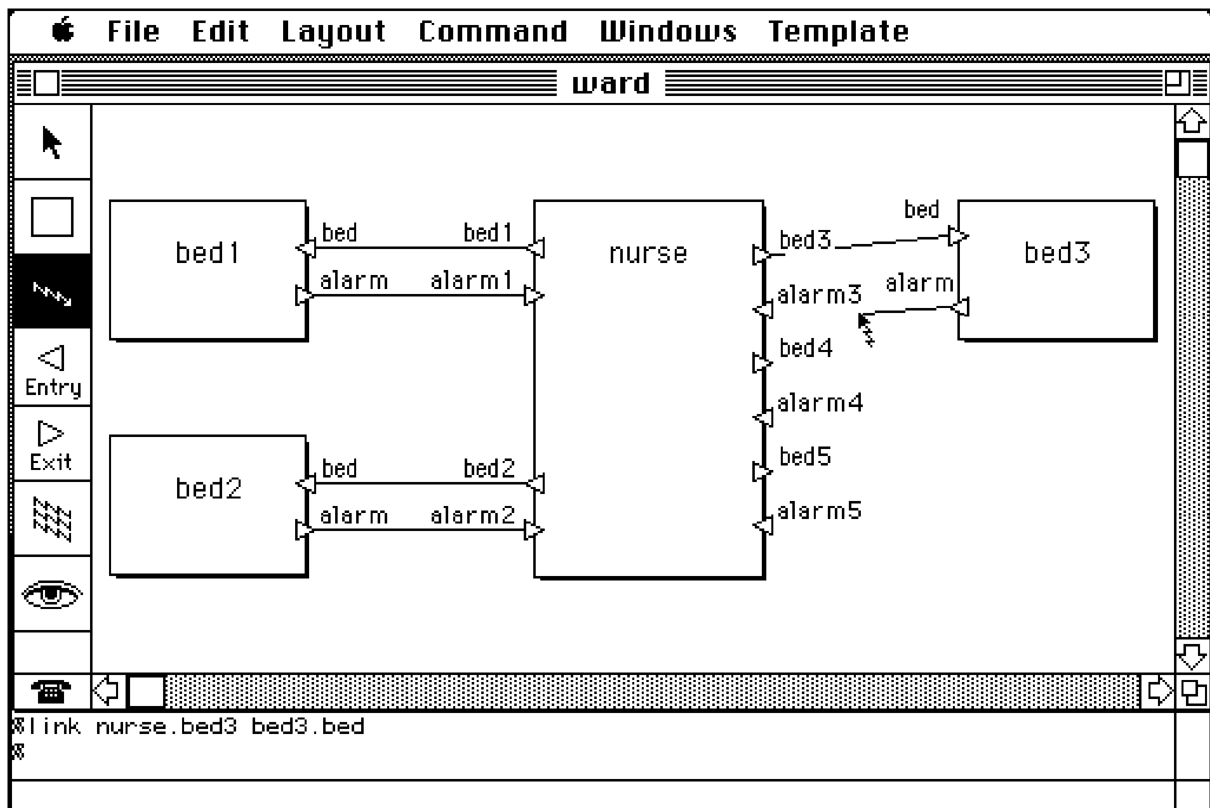


Figure 8. System Structure Monitoring and Linking via ConicDraw.

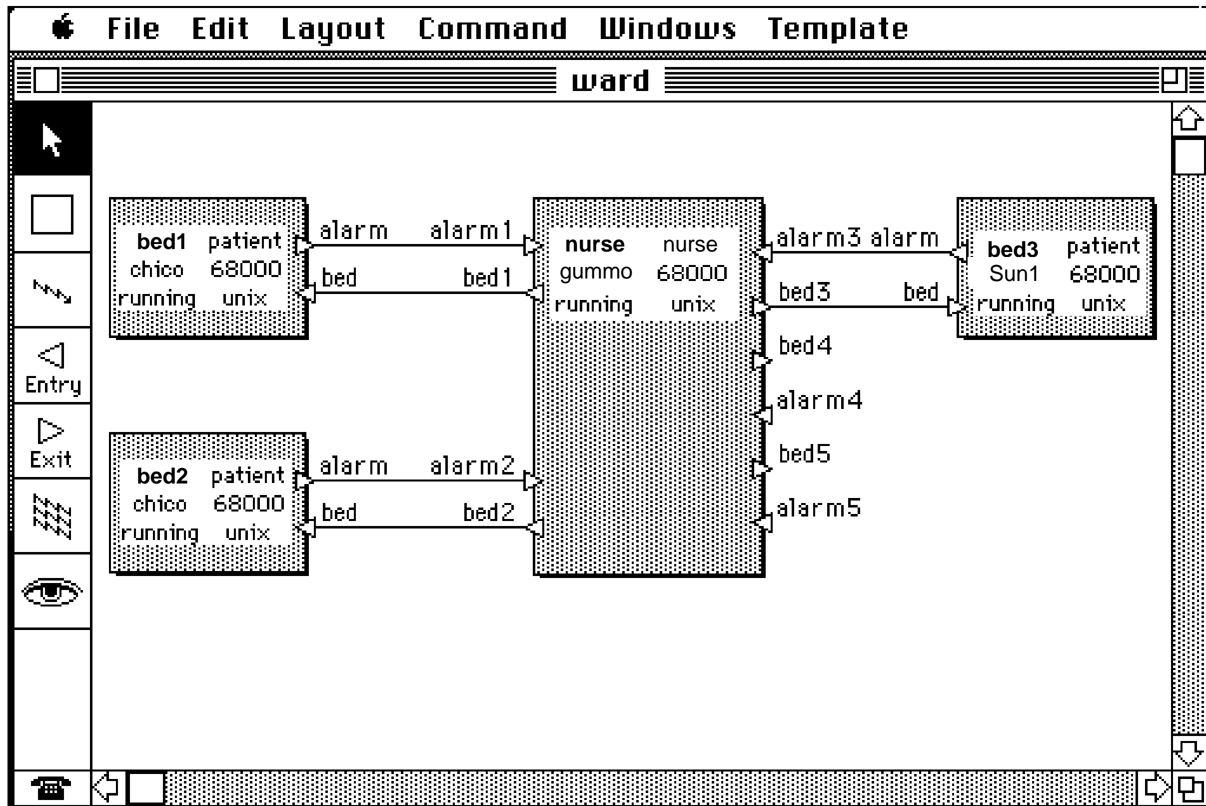


Figure 9. System Status Monitoring via ConicDraw.

For instance, the patient monitoring system being monitored in figure 8 can be extended by editing the diagram directly. These edits caused the tool to send configuration text to a configuration manager to change the actual system accordingly. Figure 8 shows a further bed, bed3, being linked into the existing system using the *link* tool in the tool palette. The link instruction generated by this graphic operation (**link** nurse.bed3 **to** bed3.bed) is shown at the bottom of the figure.

Figure 9 illustrates the status monitoring facilities available via ConicDraw, in which the current component name and type, machine location and type, and status and supporting OS are indicated for each component. A full description of ConicDraw is given in [15].

2.3 Summary and Experience

This section has briefly outlined how configuration programming is used and supported in the Conic environment. The configuration language and components both conform to the basic principles described earlier in the paper. The interested reader can find a detailed description of the Conic Configuration language in [4] and the Conic environment in [22].

The Conic environment has been in use for over 5 years. It has amply demonstrated the utility

of configuration level programming and the need for the separate configuration perspective. A number of other research projects make use of a separate configuration (or module interconnection) language (DICON [21], Durra [1], Lady [25], NETSLA [19], RNet [2], Polylith [29], STILE [32]) but few are as widely distributed and used, and as simple yet versatile as the Conic configuration language. Users of the Conic environment include a number of universities and industrial research centres in the UK, Belgium, Germany, France, Greece, Sweden, Finland, Canada, Korea, Hong Kong and Japan. Our experience at Imperial College has been very positive and confirms our belief that the structural configuration level is a useful level of abstraction for system description, construction and evolution. The provision of software tool support has been essential in the successful use of Conic. This success has encouraged us to embark on a broader and more ambitious project, based on the same principles but incorporating many more of the aspects of the development process. This ESPRIT II project, REX [30], includes work on formal specification techniques and tools for analysis and verification, on design methods and tools for recording and aiding the design process, and on configuration and dynamic reconfiguration and tools for performing the construction, reconfiguration and extension of heterogeneous distributed systems. In particular, the work on ViewPoints has also been adopted for REX.

3. VIEWPOINTS

The concept of a ViewPoint is a synthesis of the concepts of "view" and "viewpoint" in earlier work. The requirements analysis method CORE [23, 31] is based round the notion of viewpoints which is characterised as "something that does things" in the domain under consideration, akin to an agent or role. Thus the CORE viewpoint can be seen to be the particular source for *domain* decomposition. The notion of views as partial specifications and as the principal basis for incremental construction of specifications was developed in the PEACOCK [7, 8] and PRISMA [26] 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. The notion of forming “configurations” of ViewPoints is suggested by the need to provide an explicit structure for describing ViewPoint relations, and the interesting analogy between the configuration of ViewPoints used in the software process and the resulting software structure [17].

This section provides a general characterisation of ViewPoints and elaborates on the analogy with the principles, techniques and tools offered in Conic for configuration programming.

3.1 ViewPoint Definition and Characterisation

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 (figure 1) is thus a combination of the following parts 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 are elaborated and 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. For instance, if a particular ViewPoint is required to use data flow diagrams in its specification slot for a particular domain, the other slots would employ appropriate languages to specify the DFD representation, its workplan and workrecord.

ViewPoints are organised in *configurations* which are collections of related ViewPoints. In the descriptions which follow, we will show how a *method* in this setting can be viewed is a set of ViewPoint templates (types), and their relationships, together with actions governing their construction and consistency. A *specification* is viewed a configuration of ViewPoint instances.

3.2 Applying Configuration Programming Principles to ViewPoints

1. *The configuration language used for structural description should be separate from the language used for ViewPoint descriptions.*
   As before we separate the languages used for structural description from that used for the slots in primitive ViewPoints. This structural view is useful for description, comprehension and manipulation of the ViewPoints. The configuration language is *declarative*, describing what the structure is, not how it is to be constructed; that is part of the role of the workplan slot.

2. *ViewPoints should be defined as context independent types with well-defined interfaces.*
   We refer to these as *templates* (eg. figure 10). A *ViewPoint template* consists of a ViewPoint in which only the style and the work plan have been defined. Context independence means that the workplan makes no direct reference to any non-local entities ie. those mappings to other ViewPoints are indirect to permit its ready use with different ViewPoints. A ViewPoint interface describes the interaction points with other ViewPoints.

3. *Using the configuration language, complex ViewPoints should be definable as a composition of instances of ViewPoint templates.*
   The style slot of a complex ViewPoint permits the use of the configuration language to specify it as a configuration.

instances and/or their interconnections.

Modifications to the method and specification (at a large grain level) are reflected as changes to the configuration.

In order to illustrate the use of these principles and their relation to configuration programming, we now overview the configuration of ViewPoints approach, following the same structure of the description of Conic and its facilities and tools given in the previous section. The illustration is made more concrete by using the framework to overview part of the JSD Method (Jackson System Design [10]). We concentrate on the first three steps which produce an initial model of the required system and its environment:

1. Entity-Action step: identify each real world entity (object) of interest, and list the actions performed or suffered by it, and the data attributes for each action.

2. Entity-Structure step: impose an ordering on the actions of each entity (like a process) using a diagrammatic notation (as in JSP) and text.

3. Initial Model step: identify the relation between the real world entities (processes) and the system processes in a process model with connections, called an initial System Specification Diagram (SSD).

The descriptions of our framework are rough and informal so as to convey the general approach. Detailed examples are available elsewhere [6, 27].

3.3 An Analogous Configuration Framework for ViewPoints

ViewPoint Templates*:- context independent ViewPoint types (principle 2)*

As mentioned, a ViewPoint *template* elaborates only the style and workplan 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, these actions 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.

A *method* is defined as a form of configuration of a selected set of ViewPoint templates which together describe the styles and work plans to be 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. Method *use* is thus represented as a dynamically evolving configuration of ViewPoints.

The method designers are thus responsible for the definition of ViewPoint templates, while the method users are responsible for following the workplans in ViewPoint instances and for elaborating the specification in the given style.

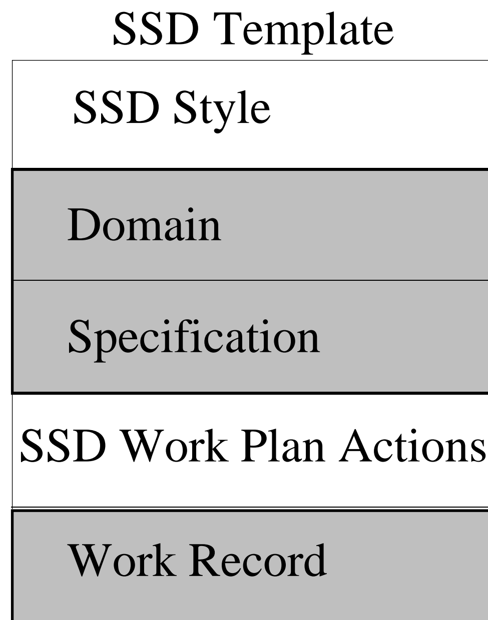


Figure 10. ViewPoint Template for System Specification Diagrams in JSD

An example of a template for the initial model step of JSD is given in figure 10. This models the real world in a style called the System Specification Diagram (fig. 11). This description provides the syntax and constraints for a "well-formed" diagram in the style. The description of the workplan actions describe how to construct a specification in SSD, together with any desired constraints on the ordering of workplan actions (eg. using pre- and post-conditions), heuristics and mappings to other ViewPoints. Other templates required for JSD include the entity-action and entity-structure templates (fig. 12). Since the structure template provides an ordering for the actions specified in the action list, we indicate this mapping by an arc. The mapping can be interpreted at the recipient in either of two manners:

1. *Either* the information received by the structure ViewPoint is used as the source of the actions to be ordered,

2. *or* the information is used by the structure ViewPoint to check consistency and completness of the actions with those of the action ViewPoint.

This dual interpretation is generally useful in that it permits the completion of the ViewPoint specifications to be performed sequentially (in which case the former interpretation may be

preferable) or concurrently (in which case the latter consistency check may be desirable). In this case, the sequential order is the most likely, but one need not enforce that constraint unless it is specifically required.

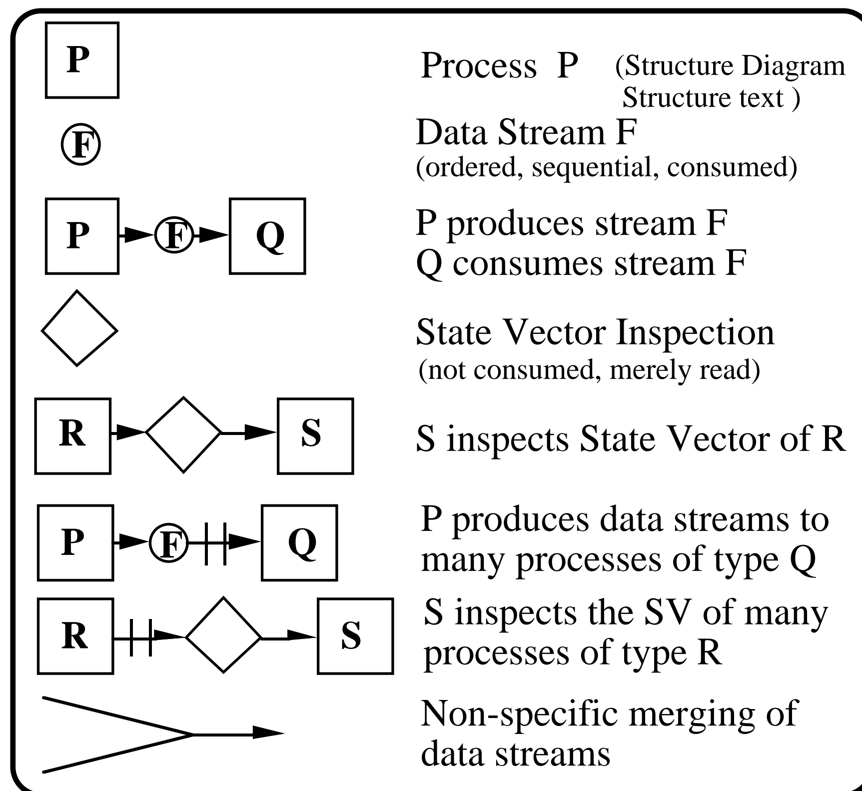


Figure 11. Outline Style for System Specification Diagrams in JSD

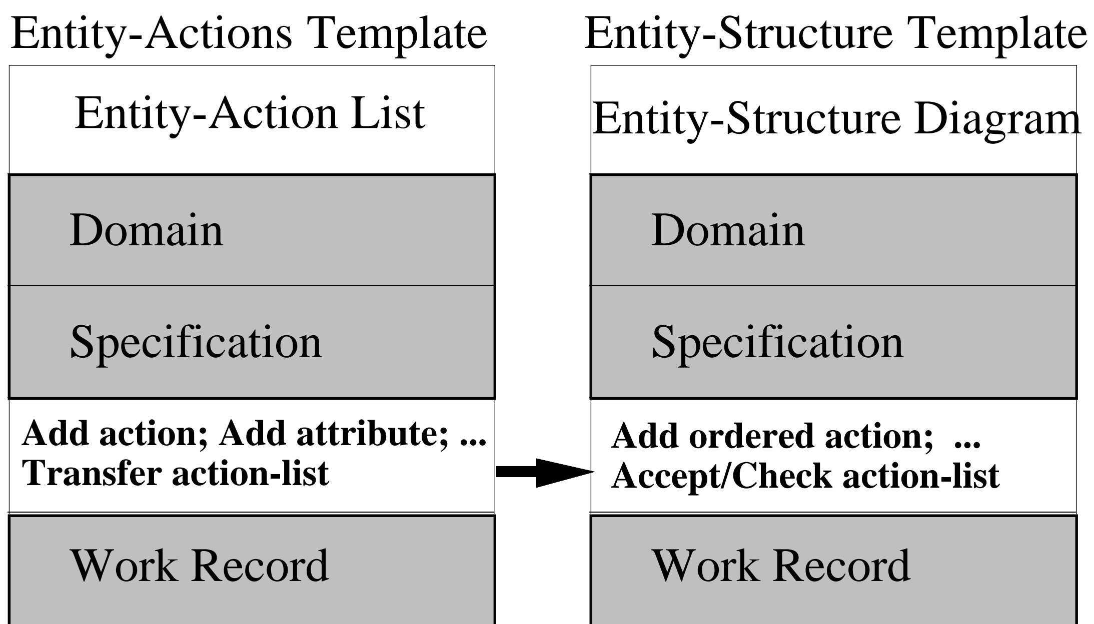


Figure 12. ViewPoint Templates for Entity-Actions and Entity-Structure

Although the principle of context independence requires that a template interface be defined, irrespective of the actual connection to other ViewPoints, we have not yet found a suitable means for the definition of template interfaces.The intention is that that part of the workplan which produces or consumes information should become the interface. One possibility is to define the mappings and their data content in separate definitions units (cf. definitions modules in Conic which can define message and port types) which are then imported.

ViewPoint Hierarchies*: - Composition of ViewPoint instances (principle 3).*

In JSD, each entity has an action-attribute list and an entity structure diagram/text. A composite template can be defined for each entity which combines these templates, and indicates the transfer of information between them and to the "interface" of the composite template (fig. 13). In this case, there is a single instance of each of the sub-templates. In general, a composite template will indicate the sub-templates and their interactions, but the number of instances may well depend on the particular application, and hence on the circumstances of instantiation of the composite template. This situation is illustrated in that, at the next level, the number of instances of entity templates will be dependent on the number of entities identified.

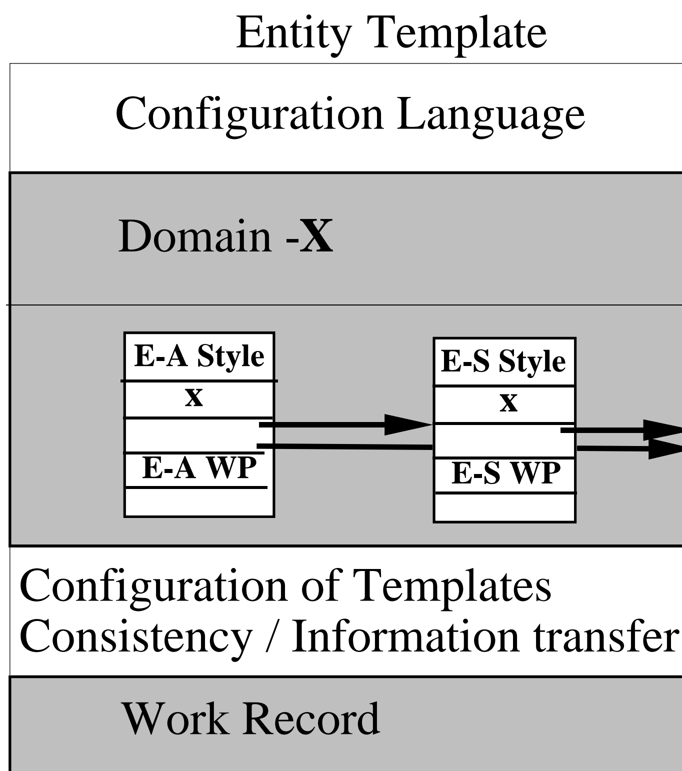


Figure 13. Composite ViewPoint Template for an Entity

Initial Model

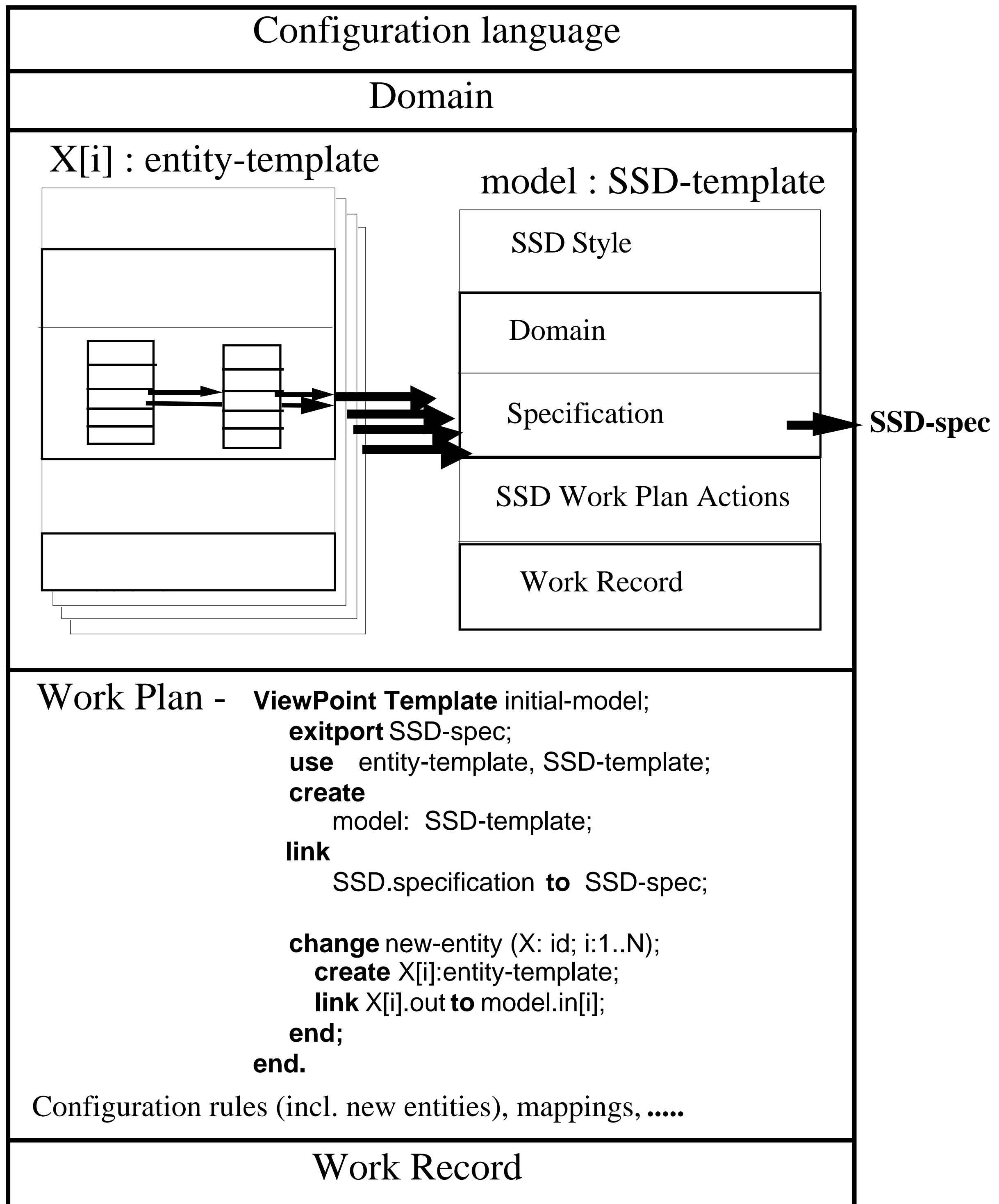


Figure 14. Initial Model Specification for JSD using ViewPoints

Constructing Specifications: *- ViewPoint instantiation and interconnection using a separate configuration language (principle 1), including dynamic configuration (principle 4).*

A *system specification* is a configuration of (consistent) specifications given in selected ViewPoint instances describing those parts of the domain which are of interest. As described above, method use is represented as a dynamically evolving configuration of ViewPoint instances. For instance, the partitioning of the application domain (into entities in JSD) provides a variable dimension to the method. For each part of the domain (entity) the method may need to create a configuration of ViewPoints as the method unfolds. This is illustrated in figure 14 for the initial model steps of JSD, together with a possible outline of the configuration description which would form part of the workplan.

We believe that the configuration and its dynamics are well described using a configuration language with the permissible changes (usually restricted to extensions) specified in the same declarative structural language. As with configuration programming, the decision as to *when* to perform configuration changes, such as ViewPoint creation and interconnection, would be described in the general workplan rather than the configuration.

Tool Support: Graphical Configuration Monitoring and Management

A particular benefit which seems to follow from the identification and encapsulation of a style (representation) and its workplan (specification method) in a single ViewPoint Template is the opportunity for tool support. We believe that individual support can 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.

In addition to individual ViewPoint tools, the configuration view of the method and specifications seems to offer the promise of practical monitoring and management tools. As with ConicDraw for configuration programming, one can envisage a graphical management tool to support monitoring the current configuration structure of an ongoing project. In addition, the status of each individual ViewPoint (reflected in its workrecord) could be monitored in order to gain insight into the current status of the project. This form of status monitoring of the partitioned parts of a method were successfully prototyped in the TARA project [13,14] for the requirements method CORE (see figure 15, which marked each part asfollows: blank- not needed for elaboration, shaded- not started, ?- started but not completed or consistent, √- completed).

Finally, the configuration management approach also offers the possibility of performing external, evolutionary adjustments and modifications to the method and development structure dynamically, while in use. Although the opportunity and facilities to perform such arbitrary adjustments does seem to be desirable, they should obviously be performed in a careful and controlled manner. The kind of change management approach adopted for configuration

programming [16] may also provide some guidance as to how to control dynamic change.

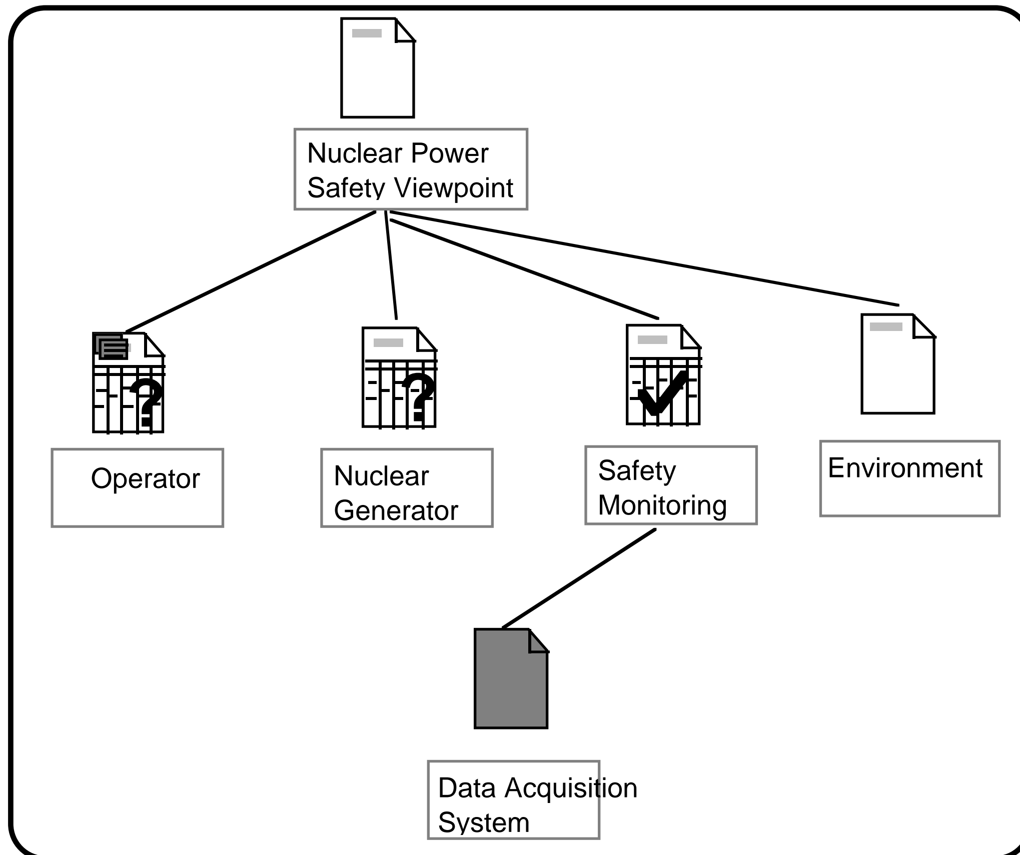


Figure 15. Status Monitoring for parts of a CORE Specification in TARA

4. CONCLUSIONS

Configuration programming, with its use of a separate configuration language, provides an excellent means for expressing system structure rather than the embedding of structural decisions in the software components themselves. The approach produces systems which are comprehensible, maintainable and amenable to change and has facilitated the provision of software tools to support system construction and management. The Conic environment for distributed programming is an exemplar for configuration programming. In this paper, we have argued that a configurable framework, analogous to that in configuration programming, can be combined with the notion of ViewPoints to provide similar benefits to the problem of method and tool integration. A vision of this configurable framework has been proposed.

The ViewPoint approach to software development advocates the use of multiple ViewPoints 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. The partitioning of knowledge exemplified in the ViewPoints approach facilitates distributed development and the use of multiple representation schemes.

Since all method steps are expressed in this common form, we believe that ViewPoints are also particularly useful in the description of integrated or mixed approaches such as those described as “multiparadigm programming” [33]. The ViewPoint approach is also strongly related to Jackson’s recent work on views and implementations [11] 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). As expected, the major issue is the expression of ViewPoint interaction.

Although the number of different kinds of relationships between arbitrary ViewPoints is theoretically enormous, we believe that, in practice, 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. It is certainly advantageous to describe the same domain using different styles to specify different aspects of behaviour.

We believe that the explicit expression of these relationships is aided by the need to consider only one formalism (style) at a time, and then to express its relationship with others. The style slot provides the required representation information while the workplan provides the place where such relations can be expressed. In addition the dual interpretation of the relationships, as information transfer or consistency check, is useful. Expressing relationships as a ViewPoint interface still requires further investigation.

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.

An interesting suggestion considering ViewPoints as active agents has been proposed by the notion of a "software development participant" in the IC~DC project [5]. There it is an active, autonomous and loosely coupled agent - in the distributed artificial intelligence style.

Current work is being conducted within the REX and SEED (Software Engineering - Engineering Design) projects. A design tool, RexDesigner, is under construction using a restricted form of ViewPoints for CDA (Constructive Design Approach [17] for distributed

programming). Other work using ViewPoints is examining methods associated with PrtNets, an extended form of Petri Nets, and CSP, and a general framework is being prototyped in Smalltalk.

Acknowledgements

Acknowledgement is made to my colleagues at Imperial College, Naranker Dulay, Anthony Finkelstein, Jeff Magee, Keng Ng, Morris Sloman and Kevin Twidle for their contribution to the configuration programming work described in this paper. Anthony Finkelstein, Michael Goedicke and Bashar Nuseibeh have contributed much to the work on ViewPoints. Finally I gratefully acknowledge the SERC under grant GE/F/04605 and the CEC in the REX Project (2080) for their financial support.

REFERENCES

[1] M.R.Barbacci, C.B.Weinstock, and J.M.Wing, "Programming at the Processor - Memory - Switch Level", Proc. of 10th IEEE Int. Conf. on Software Engineering, Singapore, April 1988.

[2] M. Coulas, G. MacEwen, G. Marquis, “RNet: A Hard Real-Time Distributed Programming System”, IEEE Transactions on Computers, C-36 (8), August 1987.

[3] F. DeRemer, H.H.Kron. "Programming-in-the-large Versus Programming-in-the-small, IEEE Trans. Software Engineering", Vol. SE-2, 2, June 1976.

[4] N. Dulay, “A Configuration Language for Distributed Programming”, Ph.D. Thesis, Imperial College, London University, 1990.

[5] A.Finkelstein and H.Fuks, “Multi-Party Specification”; Proc 5th International Workshop on Software Specification & Design; pp 185-196, IEEE CS Press.

[6] A. Finkelstein , J. Kramer, and M. Goedicke. “ ViewPoint Oriented Software Development”, Proc. of 3rd International Workshop on Software Engineering and its Applications, Toulouse, France, December 1990.

[7] M.Goedicke, W.Ditt, H.Schippers, “The ∏-Language Reference Manual”, Research Report No 295 1989, Department of Computer Science, University of Dortmund.

[8] M.Goedicke, “Paradigms of Modular Software Development” Mitchell R.J. (Ed); Managing Complexity in Software Engineering; Peter Peregrinus, 1990, England.

[9] J.A.Goguen. "Reusing and Interconnecting Software Components", IEEE Computer, (Designing for Adaptability), Vol. 19, 2, February 1986.

[10] M.A.Jackson, "System Development", Prentice Hall 1983.

[11] M.A.Jackson, “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.

[12] J.Kramer, J.Magee, "Dynamic Configuration for Distributed Systems", IEEE Transactions on Software Engineering, SE-11 (4), April 1985, pp. 424-436.

[13] 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.
[14] J. Kramer, K. Ng, C. Potts, K. Whitehead, “Tool Support for Requirements Analysis”, IEE Software Engineering Journal, Vol. 3,3, May 1988.
[15] J. Kramer, J. Magee, K. Ng, "Graphical Configuration Programming", IEEE Computer, 22(10), October 1989, 53-65.
[16] J. Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", to appear in IEEE Trans. on Software Eng., November 1990.
[17] J. Kramer, J. Magee, A. Finkelstein, “A Constructive Approach to the Design of Distributed Systems”, to be presented at the 10th Int. Conf. on Distributed Computing Systems, May 1990.
[18] J. Kramer, “Configuration Programming - A Framework fo the Development of Distributable Systems”, Proc. of IEEE Int. Conf. on Computer Systems and Software Engineering (CompEuro 90), Israel, May 1990.
[19] R.J. Leblanc and A.B. MacCabe, “The Design of a Programming Language based on a Connectivity Network”, Proc. 3rd Int. Conf. On Distributed Computing Systems, 1982.
[20] T. LeBlanc and S. Friedberg. "HPC: A model of structure and change in distributed systems". IEEE Trans. on Computers, Vol. C-34, 12, December 1985.
[21] I. Lee, N. Prywes, B. Szymanski, “Partitioning of Massive/Real-Time Programs for Parallel Processing”, in Advances in Computers, ed. M.C. Yovits, Vol.25, Academic Press 1986.
[22] J.Magee, J.Kramer, and M.Sloman, "Constructing Distributed Systems in Conic" IEEE Transactions on Software Engineering, SE-15 (6), June 1989.
[23] G.Mullery, “Acquisition - Environment”; (In) Paul, M. & Siegert, H. "Distributed Systems: Methods and Tools for Specification"; Springer Verlag LNCS 190, 1985.
[24] W.P. Myers, G.F. Myers and L.C. Constantine. "Structured design", IBM Syst. J., vol. 13, no. 2, pp. 115-139, 1974.
[25] J. Nehmer, D. Haban, F. Mattern, D. Wybranietz, D. Rombach, “Key Concepts of the INCAS Multicomputer Project”, IEEE Transactions on Software Engineering, SE-13 (8), August 1987.
[26] C.Niskier, T.Maibaum, D.Schwabe, “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.
[27] B. Nuseibeh, “ViewPoint Oriented Systems Engineering: an Interim Report and Case Study”, Internal Report, Department of Computing, Imperial College, March 1991.
[28] D.L. Parnas, "On the Design and Development of Program Families", IEEE Transactions on Software Engineering, SE-2 (1), March 1976, pp. 1-9.
[29] J. Purtilo, “A Software Interconnection Technology”, Computer Science Dept., University of Maryland, TR-2139, 1988.
[30] REX Technical Annexe, ESPRIT Project 2080, European Economic Commission, March

1989.

[31] M.Stephens, K.Whitehead, "The Analyst — A Workstation for Analysis and Design"; Proc 8th ICSE; IEEE CS Press.

[32] M. Stovsky, B. Weide, “STILE: A Graphical Design and Development Environment”, Digest Compcon Spring 87, CS Press, California.

[33] P.Zave, "An Operational Approach to Requirements Specification for Embedded Systems", IEEE Trans. on Software Engineering, SE-8 (3), 1982.