A Conceptual Graph Approach to Support Multiperspective Development Environments

Thanwadee Thanitsukkarn
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, England
E-mail: tt4@doc.ic.ac.uk

Anthony Finkelstein
Department of Computer Science, University College London
Gower Street, London WC1E 6BT, England
E-mail: A.Finkelstein@cs.ucl.ac.uk

ABSTRACT

This paper demonstrates an application of *Conceptual Graphs* (CGs) in the area of software engineering. We employ CGs as a meta-representation language to enhance consistency checking within a multiperspective development environment, i.e. one which employs and utilises a number of *ViewPoints*. We have built a ViewPoint-based prototype called the *Viewer+CG* to show such application of CGs. A ViewPoint is a loosely coupled, locally managed, self-contained object. It encapsulates representation knowledge, development knowledge, and specification knowledge of a problem domain. ViewPoints constitute partial specifications which can be independently constructed by a group of developers. Thus a complex and large-scale application can be decomposed into, and jointly managed as, a collection of ViewPoints. Partitioning development tasks and specifications in this manner necessitates a consistency checking procedure to ensure that the ViewPoints can consistently work as an 'integrated' whole. The difficulties in constructing such procedure arise from the diversity of ViewPoint representation styles. We employ CGs to provide meta-representation of ViewPoints. As CGs form a strong basis for logical reasoning, we are able to use the resulting concepts and relations from the meta-representation to establish consistency checking rules within and across ViewPoints. By abstracting a ViewPoint specification up one level to CGs, we are able to augment a ViewPoints-based environment with an automated consistency checking procedure which is independent of ViewPoint representation styles.

1. INTRODUCTION

1.1 Motivation

A single system development technique is usually not adequate to construct analysis and design models of a large-scale specification due to the complexity of the specification. Thus it is necessary to decompose such specification into small, manageable parts to which a number of appropriate development methods and tools can be applied. This, in turns, enables a specification to be described from a number of 'perspectives'. Moreover, these perspectives may be viewed in different representation styles depending on selected development methods and tools. We term a development environment that supports such variation in specification construction and representation styles as a *multiperspective development environment*.

In particular, we consider the partition and the organisation of perspectives based upon the notion of *ViewPoints* (Finkelstein, et al., 1992). Different ViewPoints provide different perspectives corresponding to actors or roles in a development process. A ViewPoint is defined as a loosely coupled, locally managed, self-contained object. It encapsulates representation knowledge, development knowledge, and specification knowledge of a problem domain. A software specification is constructed by composing a set of ViewPoints together to serve a common goal of the specification.

The use of ViewPoints resolves the difficulties, expenses and time involved in analysis and design process of large-scale applications. It enhances scalability, distribution, and incremental construction of the applications. Nonetheless, partitioning development tasks and specifications into ViewPoints necessitates a consistency checking procedure to ensure that the ViewPoints can consistently work as an 'integrated' whole.

The imposition of different representation styles of ViewPoints makes it difficult to provide a consistency checking procedure for the ViewPoints. Different ViewPoints may employ different representation styles, each of which offers a set of expressive capabilities for specifying a set of properties in the specification of a ViewPoint. Additionally, a representation style offers a set of analytical capabilities for the ViewPoint to which the style is applied. The pertinent question is how to express semantic compatibility among the styles to define consistency checking rules of ViewPoints.

We propose a solution for ViewPoint consistency checking by using *Conceptual Graphs* (Sowa, 1984) as a meta-representation language. The notion of CGs is selected for its intuitive, graphical notations, and the underlying logic. The graphs are employed to describe a Viewpoint representation style in terms of a structure of concepts and relations. As CGs form a strong basis for logical reasoning, we are able to use the resulting concepts and relations to establish consistency checking rules within and across ViewPoints.

1.2 Objective and Outline of the Paper

This paper demonstrates an application of CGs in the area of software engineering. Our proposal is to use CGs to create underlying meta-representation models of multiperspective specifications. Such models make it possible for us to implement a generic consistency checking procedure which is independent of representation styles of the specifications. We have built a prototype called the *Viewer+CG* to show such application of CGs in a ViewPoints-based framework.

Unlike the approach proposed in (Delugach, 1992a; Delugach, 1992b), we do not intend to apply CGs to interpret the entire semantics of data models in different ViewPoint representation styles. We contend that using a canonical form as such is not efficient. Firstly, it is difficult to identify a single canonical form which can express all possible representation styles. Secondly, it is difficult to implement a significant number of translation rules for the mapping between the canonical form and the representation styles. Consequently, we contend that consistency checking can be automated by using a certain level of abstraction in ViewPoint representation styles, i.e. by using a meta-representation language to describe ViewPoints. By doing so, we can resolve many of the difficulties found in using canonical forms and translation rules.

The paper is outlined as follows. Section 2 elaborates the notion of ViewPoints and its tool support. Section 3 describes the scenario which will be used to illustrate the applications of CGs in subsequent sections. Section 4 explains the use of CGs as a meta-representation language in a ViewPoints-based framework. Section 5 describes a consistency checking procedure using the resulting meta-representation and rules defined in section 4. Finally, section 5 summarises and discusses the achievement of this work.

2. VIEWPOINT-ORIENTED SOFTWARE ENGINEERING (VOSE)

ViewPoints, or generally written as *viewpoints*, are used to represent a scope of knowledge or interests of a system. The definition of viewpoints was initially employed to formalise requirement acquisition and elicitation (Mullery, 1979 ; Leite, 1989). In requirement engineering, viewpoints are seen as, for example, functions, sources and sinks of dataflows. The word *ViewPoint* distinguishes this particular notion from other multiperspective approaches. The ViewPoints concept (Finkelstein, et al., 1992) emphasises the partition of perspectives corresponding to actors or roles in a development process. This notion of ViewPoints does not constraint its use only for requirement analysis. It can also be applied to specification analysis and design by partitioning both development tasks and specifications into ViewPoints.

A ViewPoint contains three different types of knowledge, i.e. representation, development and specification knowledge. Such knowledge is assigned into five ViewPoint template slots illustrated in Figure 1.

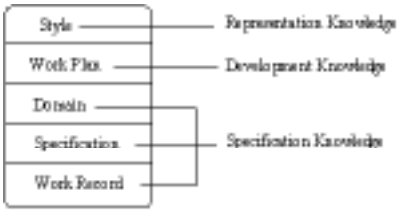

Figure 1: ViewPoint Structure

(1) The *style* slot, which specifies the notations or representation styles for a ViewPoint. The notations are instantiated to produce the specification of the ViewPoint.
(2) The *work plan* slot, which defines the following development actions.

- *Assembly actions*, which are basic editing actions to construct the specification
- *Check actions*, which are actions for consistency checking of the specification. Check actions are divided into *in-ViewPoint* and *inter-ViewPoint* actions. In-ViewPoint actions are for checking consistency within the ViewPoint in which the actions are invoked. Inter-ViewPoint actions are for checking consistency of relations among ViewPoints.
- *Guide actions*, which provide guidance to developers. The actions suggest what kinds of actions to do in the circumstances. For example, inconsistency handling actions provide guidance for possible corrections when inconsistency is encountered in a specification. (3) The *domain* slot, which is the area of concern, i.e. the problem domain, that the ViewPoint describes.

(4) The *specification* slot, which is the actual partial specification of the ViewPoint.
(5) The *work record* slot, which contains the development history, rationale and current development stage of the specification.

The realisation of the ViewPoints concept is developed as a prototype tool called the *Viewer* (Nuseibeh, 1994). In the *Viewer*, a problem is composed of a set of ViewPoints. A ViewPoint merely is a self-contained partial specification of the problem.

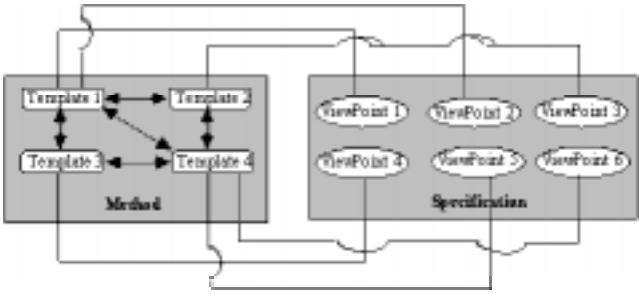

Figure 2: VOSE Architecture

The *Viewer* supports two phases of system development - the first phase is *Method Design*, the second phase is *Method Use*. *Method Design* is to specify a method in terms of ViewPoint templates. A ViewPoint template elaborates development techniques, i.e. representation styles and work plans, for a method fragment or a method. *Method Use* is to instantiate ViewPoint templates to construct a specification.

Figure 2 illustrates the architecture of VOSE in the *Viewer*. A method is described as a collection of ViewPoint templates, while a specification is described as a collection of ViewPoints. Symbolic links between a method and a specification represent the instantiation of method templates to construct ViewPoint specifications. Thus a ViewPoint is an instance of a ViewPoint template. According to the ViewPoint structure illustrated in Figure 1, the style slot and the work plan slot are instantiated in *Method Design* whereas the domain slot, the specification slot and the work record slot are instantiated in *Method Use*.

Representation styles of ViewPoints can be highly heterogeneous. Thus constructing a consistency checking procedure for the ViewPoints is not a simple task. One of possible ways to make the ViewPoints understand each other is to employ a number of translation rules. As shown in Figure 3, *n* different ViewPoint representation styles would require a significant number, *n(n-1)*, of translation rules. Another possible solution is to provide a canonical form of those representation styles. This can reduce the number of translation rules required, but it is difficult to identify a single canonical form which can express all possible representation styles. A number of approaches have been proposed to tackle the difficulties found in integrating multi-views (Meyers, 1991 ; Delugach, 1992a ; Delugach, 1992b). Each of them offers different benefits but there is no consensus about which is the most useful.

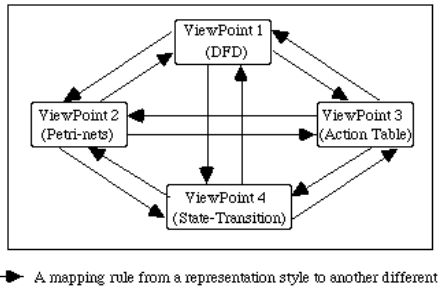

Figure 3: Translation Rules among ViewPoints with Different Representation Styles

We propose an alternative solution to construct a consistency checking procedure in a ViewPoints-based framework. We have built the *Viewer+CG* which is the modified version of the *Viewer* to support the use of CGs for ViewPoint consistency checking. We have augmented the *Viewer+CG* with CGs as two 'dimensions'. Firstly, we employ CGs as a meta-representation language to describe representation styles and consistency checking rules of ViewPoints (c.f. section 4). Secondly, we construct a consistency checking procedure based on the resulting meta-representation and rules (c.f. section 5).

3. THE SCENARIO

We present a scenario to demonstrate the application of CGs in the *Viewer+CG* environment. Our scenario demonstrates the construction of a ViewPoint template, namely *Structure and Component Identification*, and the development of a specification based on the template. The template describes the process and the representation styles of a design step in the *Constructive Design Approach* (Kramer, et al., 1990). The CDA was launched as a design technique for distributed systems exploiting the *CONIC* (Magee, et al., 1989) facilities for distributed programming. The CDA emphasises design mechanism for decomposing a system into a set of components. The principle of the CDA is that a structure of potential distributed components of a specification should be maintained throughout analysis and design process, thereby enhancing reconfiguration for the specification.

The *Structure and Component Identification* step is to identify processing components and their decomposition to produce a structural description of a distributed application. Data flows are also identified to provide the details of the structure. By applying this development step in a ViewPoints-based framework, the structure of each component can be constructed separately and jointly managed. This enables separation of concerns, thereby reducing complexity in the specification construction of the application. The scenario will show how the specification of each distributed component can be safely and independently developed through the use of a consistency checking procedure in our tool, the *Viewer+CG.* In subsequent sections, we will present a series of windows which are the results of developing the scenario in the tool.

4. CGs AS META-REPRESENTATION LANGUAGE FOR VIEWPOINTS

In this section, we demonstrate the use of CGs as a meta-representation language to describe the representation style and the work plans of the *Structure and Component Identification* template in the scenario. The process of constructing a ViewPoint template consists of two main steps. First step is to describe the representation style of the template. Second step is to specify the work plans of the template. These steps are performed in *Method Design* of the *Viewer+CG*.

4.1 Expressing Representation Styles

A ViewPoint representation style consists of a set of style components, each of which expresses description, and a graphical component of the style. To express a representation style of a ViewPoint, we firstly employ CGs as a language to constitute concepts and relations in the style. Secondly, we use the identified concepts and relations to construct meta-representation binding of the style components. This meta-representation binding forms the basis, i.e. a set of CGs, for constructing work plan definition of the ViewPoint. The construction of work plan definition will be explained in section 4.2.

4.1.1 Component Styles

4.1.1.1 Built-in Concept and Relation Type Hierarchies

To express meta-representation of ViewPoints in the *Viewer+CG*, we define built-in concept type and relation type hierarchies as in Figure 4 and Figure 5 respectively. Built-in concept type hierarchy represents the structure of concept types for style components in ViewPoint templates. Built-in relation type hierarchy represents the structure of relation types for the defined concept types. Double angle-brackets depict abstract types in the hierarchies. Abstract types classify related groups of concept and relation types in the hierarchies. They are not tangible types that can be used to construct meta-representation of ViewPoint representation styles.

Figure 4: Built-in Concept Type Hierarchy

Built-in concept types are divided into *attribute* concept types and *style* concept types. *Attribute* concept types represent attributes of style components. *Style* concept types represent style components. Each group of the types is further divided into *basic structure* types and *user-defined* types. *Basic structure* concept types are ad hoc concept types for constructing ViewPoint-based environment architecture. *User-defined* concept types are concept types which are added in the process of method construction by method designers. *Basic structure style* concept types consist of *viewpoint* concept type. *Basic structure attribute* concept types are *name*, *domain*, and *method* concept types. These attributes are initially defined for viewpoint concept type, but they can also be used for user-defined component styles.

Figure 5: Built-in Relation Type Hierarchy

Built-in relation types are also divided into *basic structure* relation types and *user-defined* relation types. Similar to *basic structure* concept types, *basic structure* relation types are for the environment construction. An *attribute* relation describes the relation between a *style* concept type and an *attribute* concept type. Examples of *attribute* relations will be illustrated in the next section. A *hold* relation describes ViewPoint ownership. For instance, a ViewPoint may hold a collection of components in its specification. The *hold* relation is also used to describe the ownership of a ViewPoint and its consistency checking rules. Examples of *hold* relations for the rules will be illustrated in Figure 18.

As defined in (Sowa, 1984), a concept of CGs can be either *generic* or *instantiated.* A generic concept is a concept without referents. An instantiated concept represents an individual concept of a particular type. For instance, [component] refers to a generic concept type component. [component: *a*] refers to an instance of concept type component that is identified by a referent *a*. A referent can be either an identifier or a variable. This definition will be applied throughout CGs dialogs in this work.

4.1.1.2 Component Styles and their Attributes

A component style is described in terms of its graphical notations and its attributes. The meta-representation language in the *Viewer+CG* captures a component style and an attribute of the style into a *style* concept type and an *attribute* concept type respectively. Figure 6 shows the component styles of the *Structure and Component Identification* template. The template expresses two types of component styles, *component* and *dataflow*.

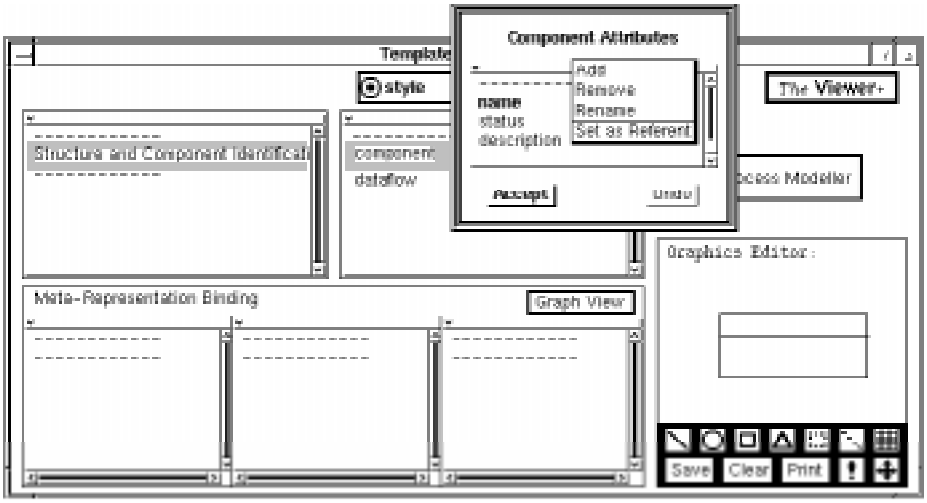

Figure 6: Representation Styles Template

A relation between a *style* concept type and its *attribute* concept type is expressed as the following general form of a CG.

[*Style Concept Type*] –> (attribute) –> [*Attribute Concept Type*].

In the *Structure and Component Identification* template, a component is identified by its name, status and description. A component status can be either *primitive* or *non-primitive*. A non-primitive component can be decomposed into subcomponents whereas a primitive component cannot be decomposed further. A dataflow is identified by its name. The pop-up menu in Figure 6 shows an attribute list of a *style* concept type, named *component.* Figure 7 illustrates the CGs representing relations between style concept types and attribute concept types in the *Structure and Component Identification* template.

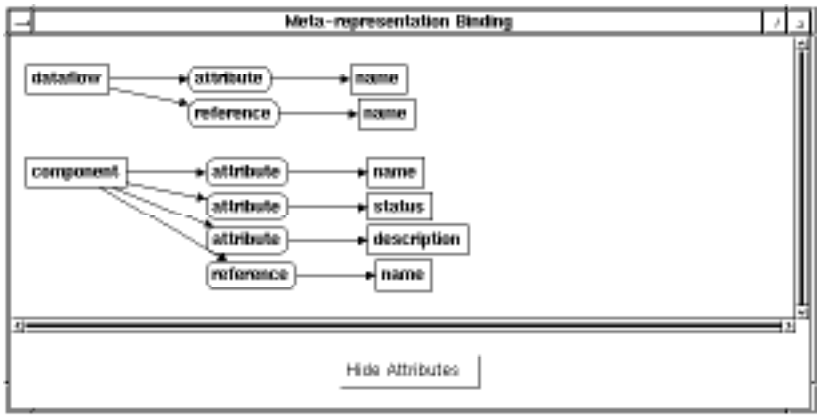

Figure 7: CGs for Component Styles and their Attributes

In the following, we employ the identified concept types to construct meta-representation binding of the *Structure and Component Identification* template.

4.1.2 Meta-Representation Binding

Meta-representation binding of a ViewPoint representation style is expressed as a collection of CGs. The graphs specify relationships among style components.

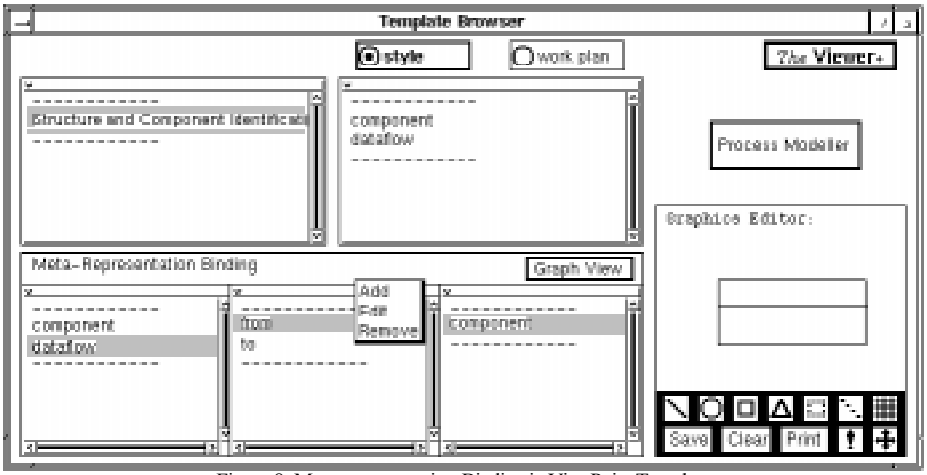

Figure 8: Meta-representation Binding in ViewPoint Template

Figure 8 depicts the meta-representation binding of the *Structure and Component Identification template*. The window for *Meta-Representation Binding* consists of three parts. The left part displays the subjects, i.e. component styles, of the binding. The middle part displays the relevant relations of the subjects. The right part displays the relevant

objects, i.e. component styles, of the relations. A binding clause is read horizontally. For instance, the highlighted meta-presentation binding clause in the figure illustrates a binding between a *dataflow* concept type and a *component* concept type by a relation type named *from*. The clause specifies that a dataflow can be sent from a component.

Generally, a binding clause is defined as a conceptual relation from a style component to another style component. The clause is written as the following general form of a CG.

[*Style Concept Type*] –> (*Relation Type*) –> [*Style Concept Type*].

Each binding clause is constructed by using the dialog illustrated in Figure 9. The pop-up menu in the figure shows a list of concept types which can be used to construct meta-representation binding of the *Structure and Component Identification* template. These concept types are identified by the steps in previous section.

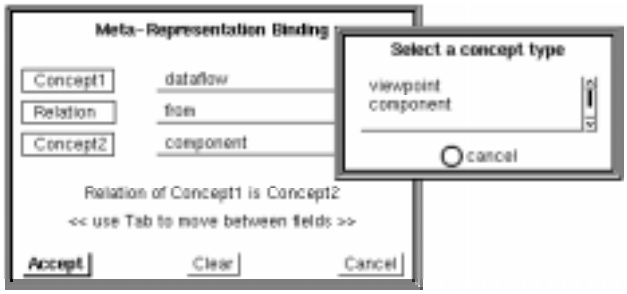

Figure 9: Meta-representation Binding Dialog

Figure 10 depicts the CGs which represent the meta-representation binding of the *Structure and Component Identification* template. A relation *from* between a *component* concept type and a *dataflow* concept type represents the dataflow sending from the component. A relation *to* between a *component* concept type and a *dataflow* concept type represents the dataflow sending to the component. A relation *decomposition* between a *component* concept type and a *viewpoint* concept type represents the ViewPoint as a decomposition of the component.

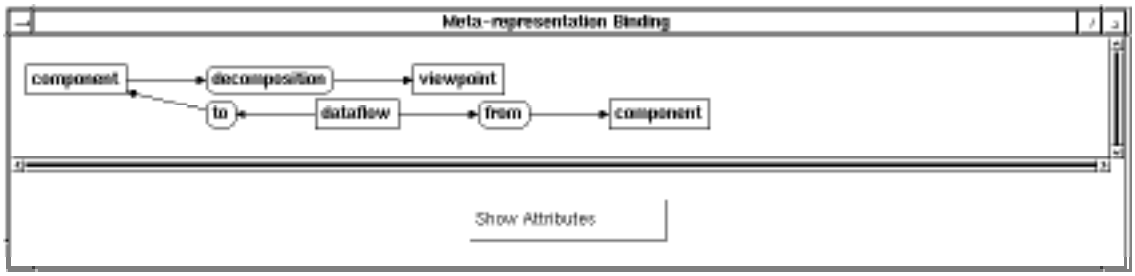

Figure 10: Meta-representation Binding (without component attributes)

Figure 11 depicts the complete CGs of the representation style of the *Structure and Component Identification* template. The graphs when instantiated produce an abstract view, i.e. an instance of meta-representation, of a ViewPoint specification using the template.

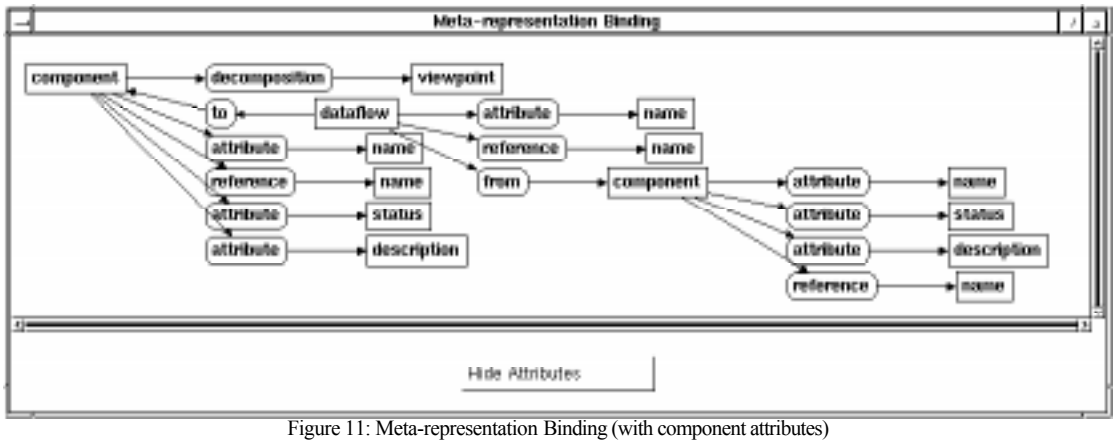

Figure 11: Meta-representation Binding (with component attributes)

In the following, we use the resulting CGs from the meta-representation binding of the *Structure and Component Identification* template to establish assembly actions and consistency checking rules of the template.

4.2 Expressing Work Plans

ViewPoint work plans describe process knowledge, i.e. a series of actions and rules that are required for the construction of ViewPoint specifications. Here CGs are employed in two aspects. Firstly, we interpret an assembly action as a sequence of CGs operations. The operations are employed to implicitly transform a ViewPoint specification into CGs through the construction of the specification in *Method Use* (c.f. section 5.1). Secondly, we establish consistency checking rules of ViewPoints by using CGs.

4.2.1 Assembly Actions

An assembly action is interpreted as a sequence of graph operations. The sequence provides guidance to method designers to implement the action accordingly.

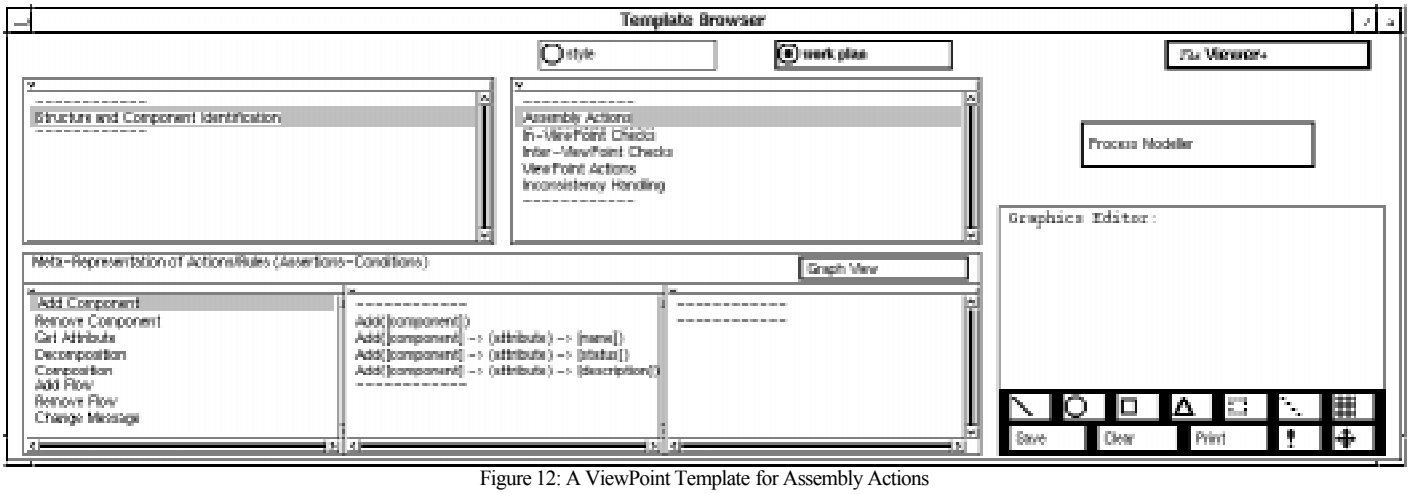

Figure 12: A ViewPoint Template for Assembly Actions

We specify the following types of graph operations to constitute assembly actions.
(1) Add operation is to add a CG.
(2) Delete operation is to delete a CG.
(3) Retrieve operation is to perform a query on a CG.
(4) Update operation is to retrieve a CG, and save the changes which might be made.

Figure 12 illustrates a ViewPoint template for assembly actions of the *Structure and Component Identification* step. The left part of the *Meta-Representation of Actions/ Rules* window displays a list of the actions. The middle part of the window displays a list of graph operations for a selected action. From the figure, an action *Add Component* is interpreted as a sequence of graph operations for adding a *component* concept, and the *attribute* concepts of the *component* concept.

4.2.2 Consistency Checking Rules

As a result of expressing representation styles as CGs, we are able to establish consistency checking rules using the graphs. We define a general form of the rules as follows.

Conditions –> Assertions

where *Conditions* and *Assertions* are conjunctions of CGs. The *Assertions* must be hold if the *Conditions* are hold.

In this scenario, we specify two consistency checking rules for the development of a specification using the *Structure and Component Identification* template. Firstly, we specify the *Invalid Component Status* rule which enforces that only components with non-primitive status can be decomposed. Secondly, we specify the *Input Decomposition Checking* rule to verify the consistency of input flows between a component and its decomposition. The first rule is defined as an in-ViewPoint rule as it requires only information within the ViewPoint in which the rule is invoked. The second rule is defined as an inter-ViewPoint rule as the rule requires information across ViewPoints. The following sections elaborate how the rules are constructed.

4.2.2.1 In-ViewPoint Rules

Consider the definition of the *Invalid Component Status* rule. The rule states that for any component which has a decomposition, the status of that component should be specified as *Non-Primitive*.

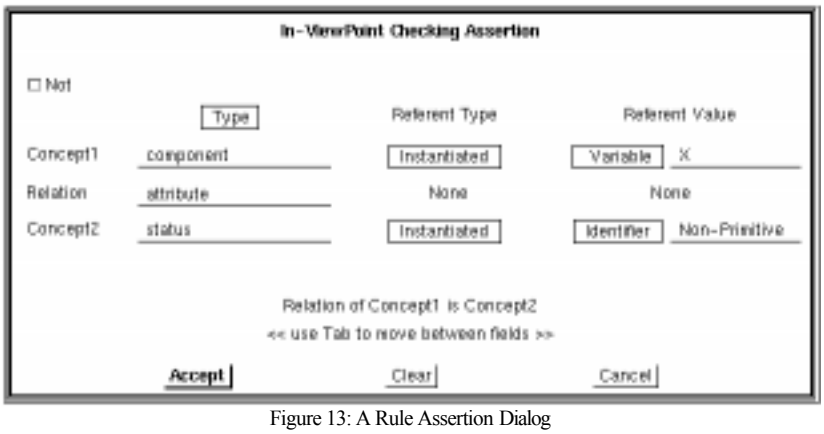

Figure 13: A Rule Assertion Dialog

Figure 14 depicts a dialog representing the definition of the *Invalid Component Status* rule. The assertion clause in the figure is constructed by using the dialog in Figure 13.

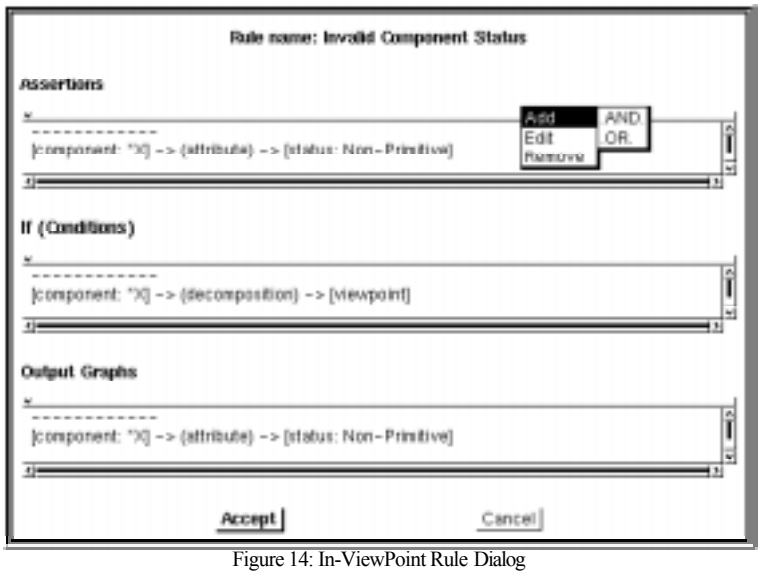

Figure 14: In-ViewPoint Rule Dialog

To compose an in-ViewPoint checking rule, a method designer is required to define three types of CGs clauses. The first type represents the conditions of the rule. The second type represents the assertions of the rule. The third type defines output graphs of the rule. Output graphs illustrate missing assertions that we want to identify as a result of consistency checking procedure. A possible set of output graphs is therefore derived from assertion clauses. The use of output graphs will be illustrated in section 5.2.

Figure 15 illustrates a ViewPoint template for in-ViewPoint checking rules. The left part of the *Meta-Representation Actions/Rules* window displays a list of in-ViewPoint rule names. The middle part of the window displays assertion clauses of a selected rule. The right part of the window displays condition clauses of the rule. In this example, the *Graph View* of the selected rule, i.e. the *Invalid Component Status* rule, will be expanded to Figure 16. Figure 16 depicts CGs representing the rule. The top part of the window in the figure illustrates CGs of the rule's assertions, while the bottom part of the window shows CGs of the rule's conditions.

Figure 15: In-ViewPoint Rule Template

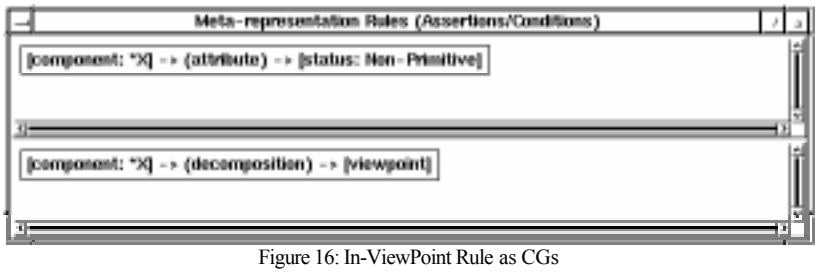

Figure 16: In-ViewPoint Rule as CGs

4.2.2.2 Inter-ViewPoint Rules

Consider the definition of the *Input Decomposition Checking* rule. The rule states that if a component has a decomposition ViewPoint, the inputs of the component must be delegated to the ViewPoint. This rule ensures a consistent *parent-child* relation between the component and its decomposition. Relations between ViewPoints are established from the ViewPoint specifications which hold assertions and conditions of the rule.

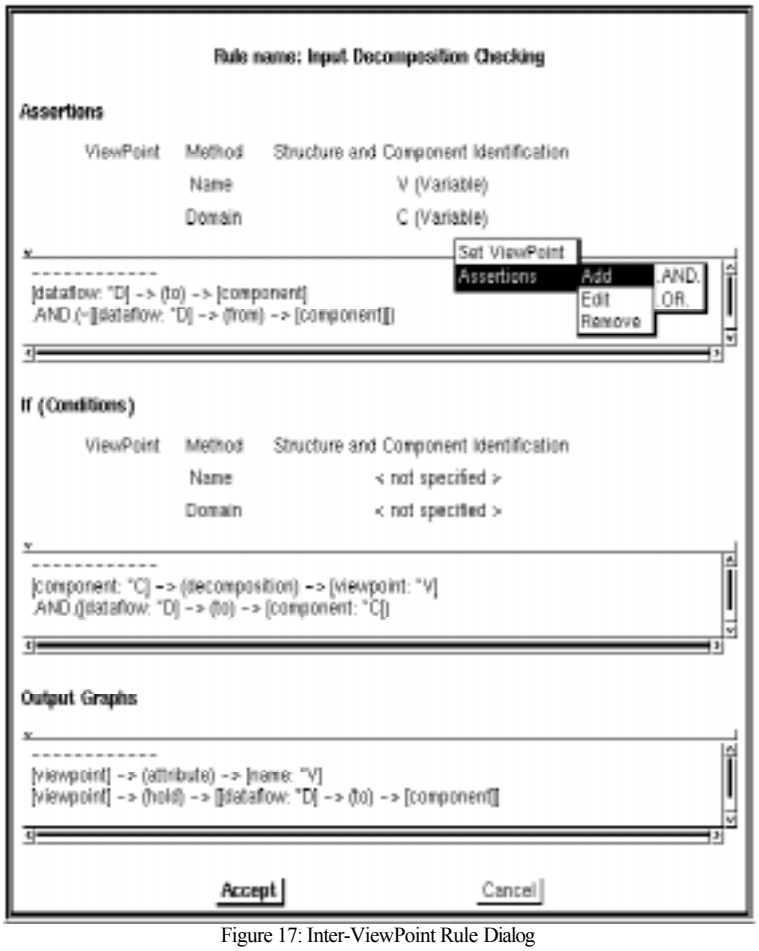

Figure 17: Inter-ViewPoint Rule Dialog

The process of composing an inter-ViewPoint rule is similar to that of an in-ViewPoint rule. Figure 17 depicts the dialog for constructing the definition of the *Input Decomposition Checking* rule. Figure 18 illustrates CGs of the rule.

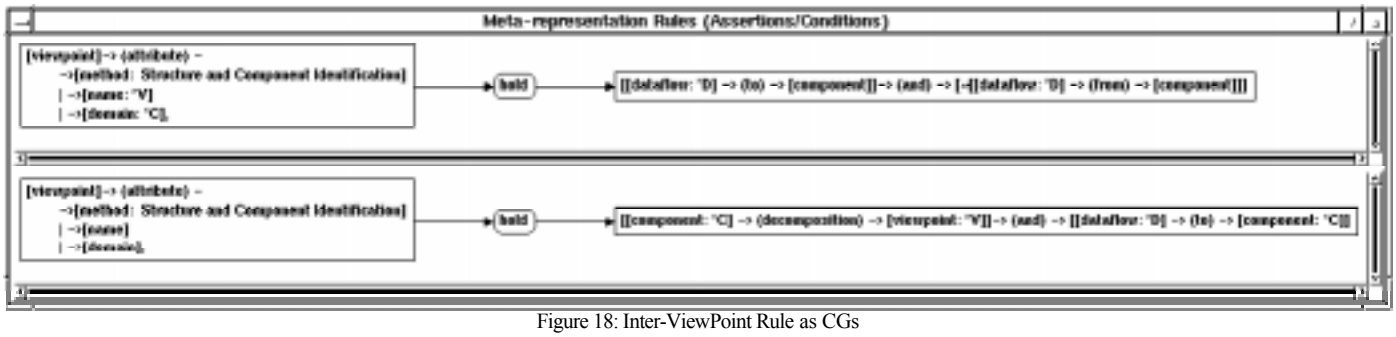

Figure 18: Inter-ViewPoint Rule as CGs

In the next section, we will illustrate how to use the CGs definition of representation styles and work plans to automate the process of consistency checking in ViewPoints.

5. CONSISTENCY CHECKING PROCEDURE

In this section, we employ the template definition of the *Structure and Component Identification* step (c.f. section 4) to construct two ViewPoints representing the main system and its decomposition of a distributed application. Then we perform consistency checking between the ViewPoints. This part of the scenario is carried out in *Method Use* of the *Viewer+CG*. We do not intend to provide details of *Method Use* functions here. Readers should refer to (Nuseibeh, 1994) for further explanation of *Method Use*.

5.1 Transformation of ViewPoints Specification to CGs

Figure 19 illustrates a ViewPoint *VP1* representing main processing of a distributed application. In this scenario, the application contains two main processing components, *a* and *b*. The data flows, named *in* and *out*, *ab*, in the figure represent the interaction between the components and external entities and the interaction among the components themselves.

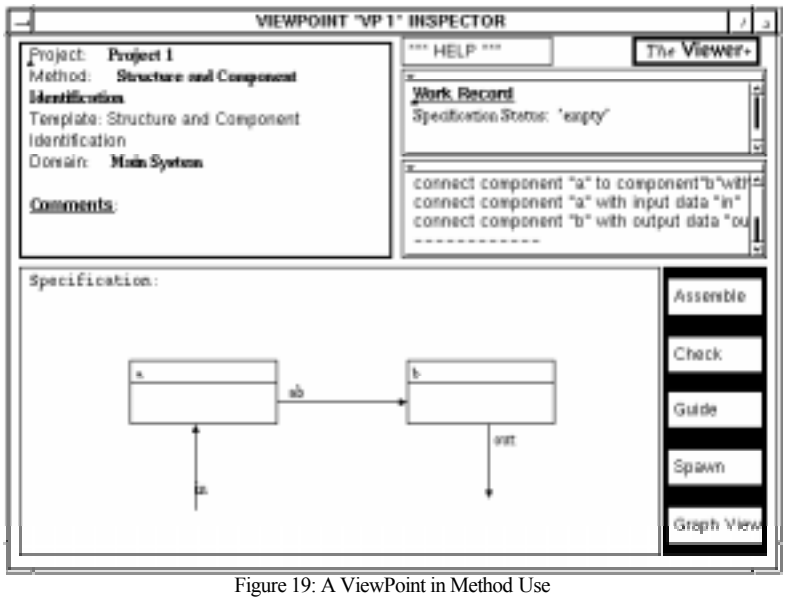

Figure 19: A ViewPoint in Method Use

To construct the specification of *VP1*, we perform the following steps.

- Firstly, we perform an assembly action to add a primitive component *a* to the ViewPoint specification. The action invokes a dialog to request a user to fill in the values of the component's attributes. A sequence of graph operations for an *Add Component* action (c.f. Figure 12) is then performed to construct the specification. The CGs presented in Figure 20 is the result of the instantiation of the ViewPoint meta-representation (c.f. Figure 11) for this action.

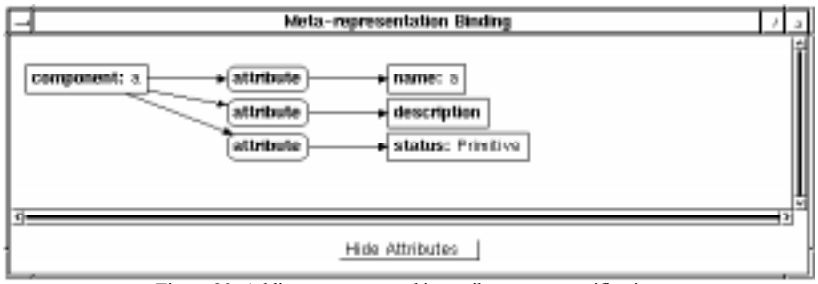

Figure 20: Adding a concept and its attributes to a specification

- Secondly, we add a primitive component *b* to the specification. This triggers a similar sequence of graph operations as the one in the first step. Then we connect a dataflow *ab* from the component *a* to the component *b*. The dataflow connection triggers three graph operations. The first operation adds a concept type dataflow named *ab*. The second operation adds a relation type *from* that links the dataflow concept *ab* to the component concept *a*. The last operation adds a relation type *to* from the dataflow concept *ab* to the component concept *b*. Figure 21 illustrates CGs which are the results of the operations.

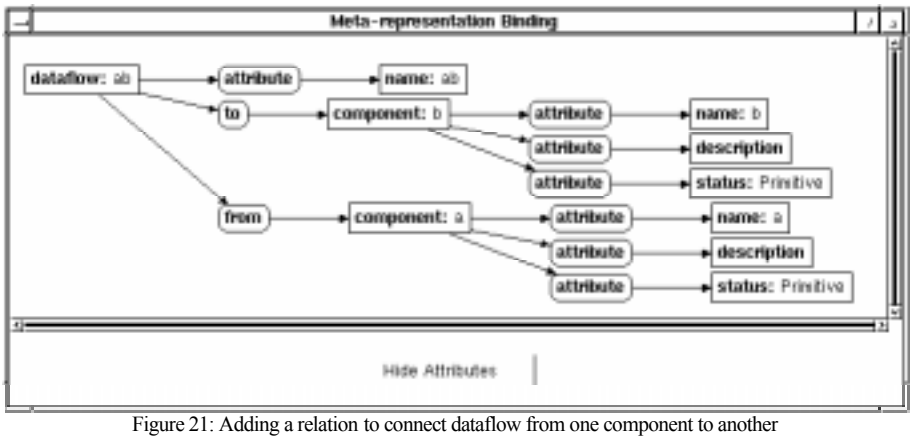

Figure 21: Adding a relation to connect dataflow from one component to another

- In the next step, we connect the dataflow *in* as an input of the component *a*. After that, we connect the dataflow *out* as an output of the component *b*.

- Finally, we decompose the component *b* into a ViewPoint *VP2*.

Figure 22 depicts the complete transformation of the ViewPoint specification in Figure 19, including the attribute declaration of style components, to CGs.

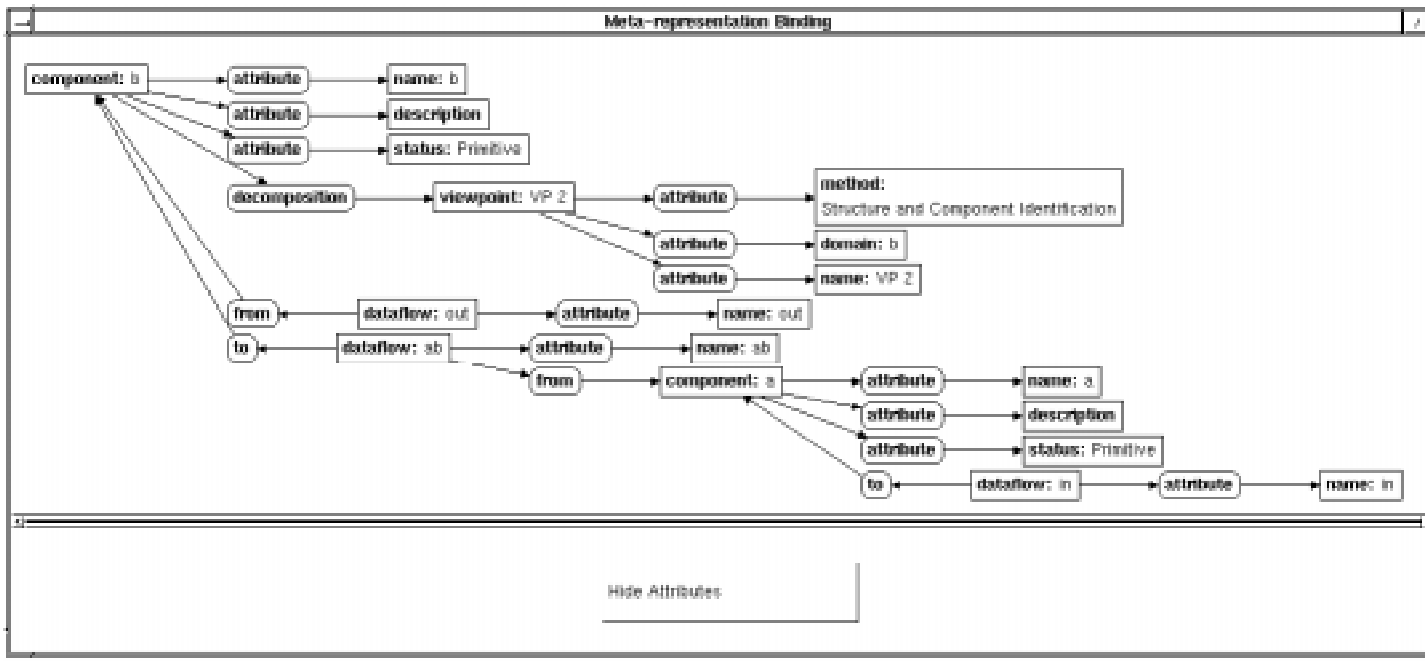

Figure 22: CGs Transformation of a ViewPoint Specification, *VP1*

We further develop the ViewPoint *VP2* which is the decomposition of the component *b*. The component *b* is decomposed into two components, namely *c* and *d*. The dataflows for the decomposition are illustrated as in Figure 23.

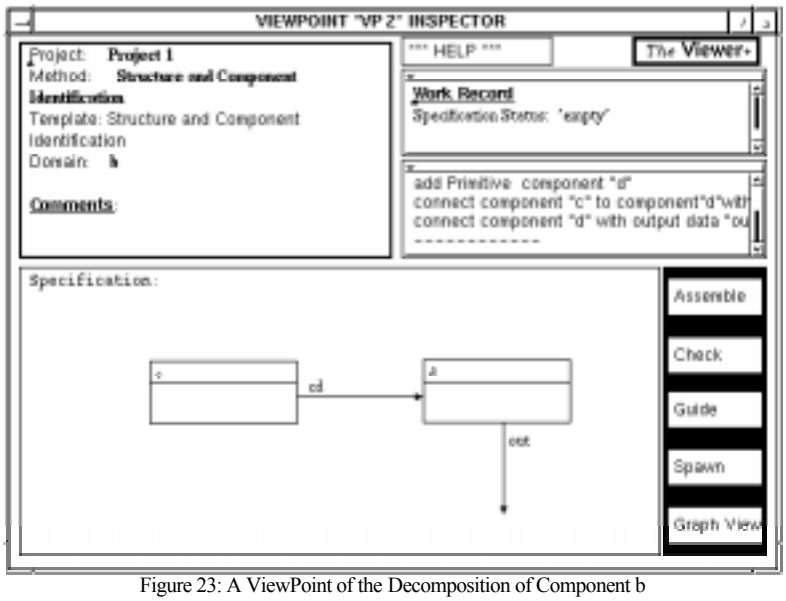

Figure 23: A ViewPoint of the Decomposition of Component b

Figure 24 illustrates the transformation of the specification of *VP2* into CGs.

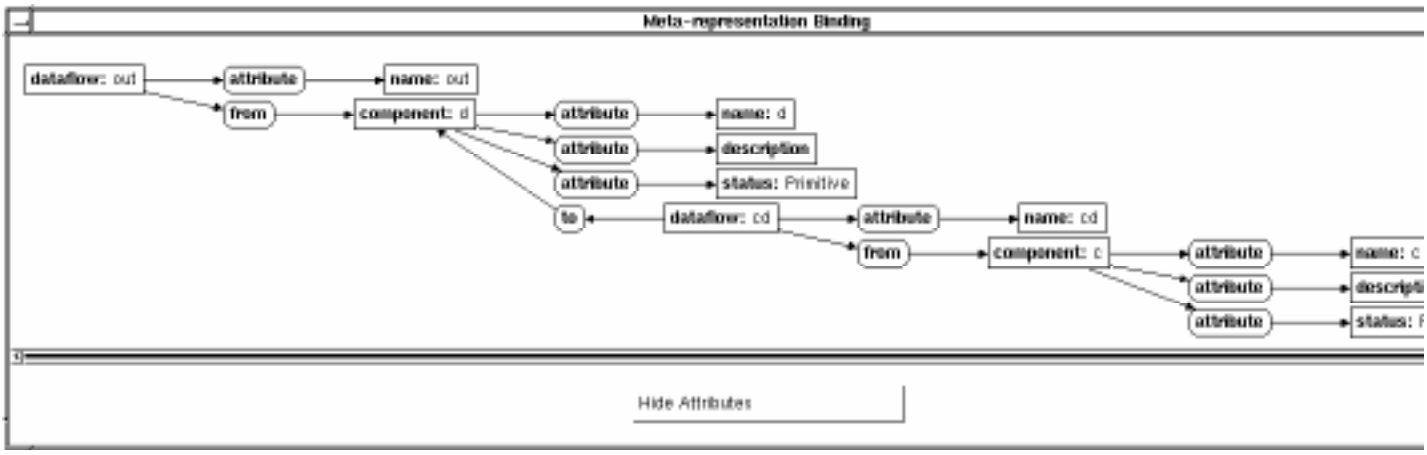

Figure 24: CGs of the decomposition ViewPoint VP2

In a multiperspective development environment, the ViewPoints *VP1* and *VP2* may be independently constructed by two different developers. As a consequence, the relations between the ViewPoints need to be verified by performing consistency checking between the ViewPoints.

5.2 Reasoning

To ensure consistency of the specification that we have built in section 5.1, we perform consistency checking according to the rules defined in section 4.2.2. Firstly, we invoke in-ViewPoint checking at the ViewPoint *VP1*. Then we invoke inter-ViewPoint checking to validate the relations between *VP1* and *VP2*.

In-ViewPoint checking process performs the logical reasoning between selected rules and the CGs of the ViewPoint specification in which the check is invoked. Figure 25 illustrates the result of in-ViewPoint Checking on *VP1*. The checking process performs the logical reasoning between the CGs of *VP1* (c.f. Figure 22) and the CGs representing the *Invalid Component Status* rule (c.f. Figure 16). The result in Figure 25 shows that the status attribute of component *b* is invalid. Component *b* was decomposed into ViewPoint *VP2*, therefore, the status of the component should be non-primitive. The top right part of the window in the figure shows a possible inconsistency handling action by means of CGs for this rule. As described earlier, the inconsistency action is instantiated from output graphs of the rule. The graphs can be transformed into ViewPoint actions. The transformation, however, is out of the scope of this work

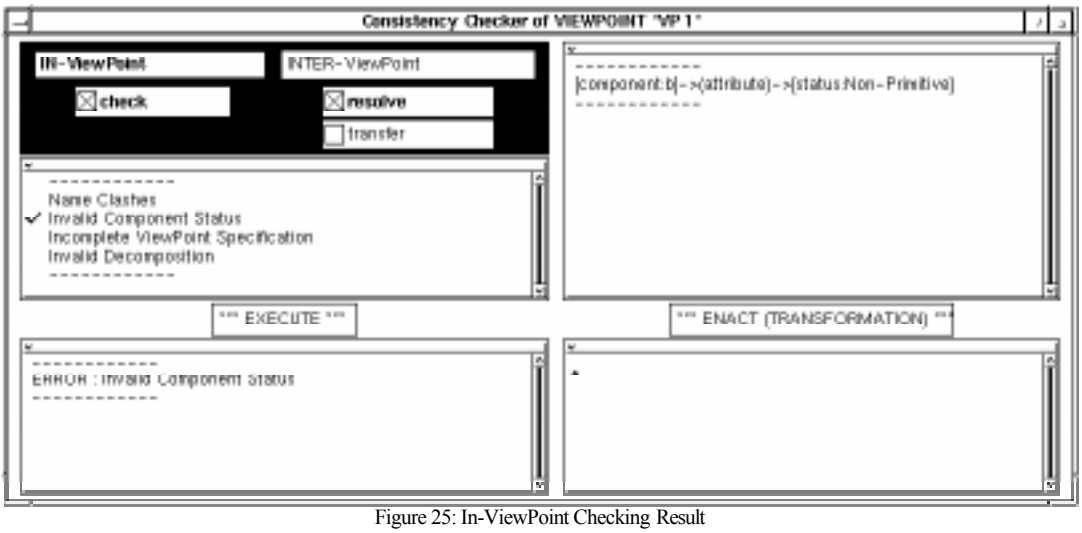

Figure 25: In-ViewPoint Checking Result

Next, consider that we perform inter-ViewPoint checking, in particular *Input Decomposition Checking* rule, at *VP1*. The process of inter-ViewPoint checking is similar to that of in-ViewPoint checking, but it performs the checking on two or more ViewPoint specifications. The number of the specifications depends on the instantiation of ViewPoint relations according to the inter-ViewPoint rules during the execution of the check.

To perform the *Input Decomposition Checking* rule, the consistency checking procedure firstly inputs the CGs of *VP1* and *VP2* (c.f. Figure 22 and Figure 24, respectively) as its assertions. Then it performs logical reasoning on the assertions and the CGs of the *Input Decomposition Checking* rule (c.f. Figure 18).

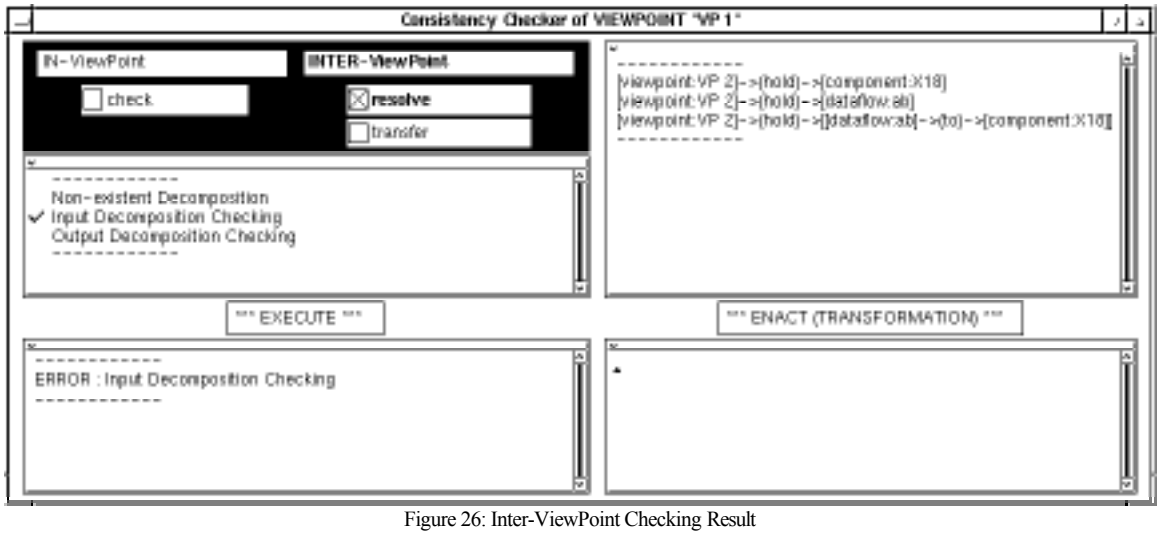

Figure 26: Inter-ViewPoint Checking Result

Figure 26 shows the result of the inter-ViewPoint checking on *VP1*. In this scenario, the input *ab* is missing from *VP2*. Therefore, there is an error of input decomposition checking between *VP1* and *VP2*. As seen in the figure, the resulting output graphs of this check imply that dataflow *ab* should be defined as an input to *a* component in the ViewPoint *VP2*. Thus an inconsistency handling action can be carried out by adding a dataflow *ab* to one of the components in *VP2*.

6. SUMMARY AND DISCUSSION

In this paper, we have presented a way of conducting a consistency checking procedure in a ViewPoints-based framework. We avoid the use of a single exhaustive canonical form to translate the entire semantic sets of various representational styles. Instead, we provide an abstract way to capture semantic compatibility among ViewPoints by using CGs as a meta-representation language. The CGs are employed for the following tasks:

(1) To visualise concepts and relations that are expressed by a method or a method fragment.
(2) To establish meta-representation binding of the concepts and the relations.
(3) To interpret an assembly action as a sequence of CGs operations. The operations are employed to implicitly transformed a ViewPoint specification into CGs.
(4) To establish in-ViewPoint rules and inter-ViewPoint rules of ViewPoints by using the resulting concepts and relations in (1).

(5) To perform logical reasoning on CGs transformation of ViewPoints specifications and the rules.

Consistency checking and integration of multiple views have been tackled by a number of researchers. The work of (Meyers, 1993) applied *Semantic Program Graphs* (SPGs) for representing software systems in a multiple-viewed development environment. However, the work is concerned with multiple views of program development techniques rather than those of analysis and design of specifications. Compared with CGs, the notion of SPGs is less appropriate to be used as a meta-representation language. Some SPGs notations, such as ones for scopes, name binding, and looping structure, are not encountered in the stages of the development process in a ViewPoints-based framework.

Although descriptive and logic-based consistency checking rules for ViewPoints have been proposed (Nuseibeh et al., 1993; Finkelstein, et al., 1994; Hunter, and Nuseibeh 1995), there is yet to be a generalized procedural mechanism to do so. CGs can be used as a bridge between the logic-based rules and the descriptive rules. The graphs provide intuitive notations to describe consistency checking rules and expressive power to transform the rules for logical reasoning.

The work of (Delugach, 1992a; Delugach, 1992b) presented the abilities of CGs to translate various representation styles of ViewPoints. However, it can be exhaustive to implement a number of translation rules using a canonical form as such for the ViewPoints. Alternatively, we employ CGs as a meta-representation language to describe ViewPoints. The resulting ViewPoint meta-representation forms a basis to establish consistency checking rules for different ViewPoint representation style. By doing so, we can resolve many of the difficulties found in using canonical forms and translation rules. The use of meta-representation enables method designers to establish semantic compatibility among different ViewPoint representation styles without implementing the translation rules of the styles.

By abstracting a ViewPoint specification up one level to CGs, we are able to augment the ViewPoints-based framework with an automated consistency checking procedure which is independent of ViewPoint representation styles. In addition, CGs provide an efficient foundation for communication between ViewPoints in a distributed environment as they can be served as a knowledge interchange format. An ongoing research of this work is to employ the *Viewer+CG* to further construct a configuration-oriented development methodology. The methodology is the synthesis of *Constructive Design Approach* (Kramer et al., 1990) and object-oriented *UML*-based notations (Booch, et al., 1997). Together with the consistency checking procedure in the *Viewer+CG* that we have presented, the methodology enables a team of software developers to safely and expediently decompose a distributed application, the analysis and design tasks for the application into different ViewPoints of configurable distributed components. By making the development process of a distributed application itself distributed, this resolves the difficulties, expenses and time involved in analysis and design process of the application.

6. REFERENCES

Booch, G., Jacobson, I., and Rumbaugh, J. (1997). The Unified Modeling Language for Object-Oriented Development, Rational Software Cooperation, September, Documentation Set Version 1.1.

Delugach, H. S. (1992a). Analysing Multiple Views of Software Requirements, in T. E. Nagle, J. A. Nagle, L. L. Gerholz, and P. W. Eklund (Eds), Conceptual Structures: Current Research and Practice, Ellis Horwood.

Delugach, H. S. (1992b). Specifying Multiple-Viewed Software Requirements with Conceptual Graphs, Journal of System Software, Vol. 19, pp. 207-224.

Finkelstein, A., Kramer, J., Nuseibeh B., Finkelstein, L., and Goedicke, M. (1992). Viewpoints: A Framework for Integrating Multiple Perspectives in System Development, International Journal of Software Engineering and Knowledge Engineering, Vol. 2(1), World Scientific Publishing Co., March, pp. 31-58.

Finkelstein, A., Gabbay, D., Hunter, A., Kramer, J., and Nuseibeh, B. (1994). Inconsistency Handling in Multi-Perspective Specifications, IEEE Transactions on Software Engineering, Vol. 20(8), December, pp. 569-578.

Hunter, A., and Nuseibeh, B. (1995). Managing Inconsistent Specifications: Reasoning, Analysis and Action, Technical Report Number 95/15, Department of Computing, Imperial College, London, UK, October.

Kramer, J., Magee J., and Finkelstein, A. (1990). A Constructive Approach to the Design of Distributed Systems, Proceedings of the 10th IEEE ICDCS, Paris, May.

Leite, J. C. S. P. (1989). Viewpoint analysis: A Case Study, Proceedings of 5th International Workshop on Software Specification and Design, ACM SIGSOFT, Vol. 14, Pittsburgh, Pennsylvania, USA, May 19-20, pp. 111-119.

Magee, J., Kramer, J., and Sloman, M. (1989). Constructing Distributed Systems in CONIC, IEEE TSE 15, June 6.

Meyers, S. D. (1991). Difficulties in Integrating Multiview Development Systems, IEEE Software, January, pp. 49-57.

Meyers, S. D. (1993). Representing Software Systems in Multiple-View Development Environments, PhD. Thesis, Department of Computer Science, Brown University, May.

Mullery, G. (1979). CORE - a method for controlled requirements expressions, Proceedings of the 4th International Conference on Software Engineering (ICSE-4), IEEE Computer Society Press, pp. 126-135.

Nuseibeh, B., Kramer, J., and Finkelstein, A. (1993). Expressing the Relationships Between Multiple Views in Requirements Specification, Proceedings of the 15th International Conference on Software Engineering, IEEE CS Press, Baltimore, USA, May.

Nuseibeh, B. (1994). A Multi-Perspective Framework for Method Integration, PhD. Thesis, Department of Computing, Imperial College, University of London.

Sowa, J. F. (1984). Conceptual Structures: Information Processing in Mind and Machine, Addison-Wesley, Reading, MA.