Published in Software - Practice and Experience Vol. 20, No. 6., Wiley

Supporting flexible consistency management via discrete change description propagation

JOHN C. GRUNDY
Department of Computer Science, University of Waikato, Private Bag 3105, Hamilton, New Zealand

JOHN G. HOSKING AND WARWICK B. MUGRIDGE
Department of Computer Science, University of Auckland, Private Bag, Auckland, New Zealand

SUMMARY

A new software architecture for supporting inter-object consistency management is described. Objects with interdependent data values are kept consistent by propagating descriptions of object state changes along inter-object relationships. Response to and storage of these change descriptions supports the implementation of consistency management techniques in a more homogeneous way than existing models. Such techniques include efficient attribute recalculation and constraint schemes, multiple view consistency, and undo-redo, versioning and cooperative work facilities. Applications of the new architecture to user interface, graphical editor and programming environment construction are described.

KEY WORDS consistency management change propagation constraints attribute grammars multiple views

1. INTRODUCTION

Graphical user interfaces, editing tools and programming environments require mechanisms to keep interdependent data values consistent under change[1, 2, 3]. These consistency management mechanisms should support the recalculation and constraint of values when related data changes, multiple textual and graphical views (representations) of data, undo/redo of editing changes, and version control and cooperative work facilities. As these kinds of consistency management are common, a reusable software architecture for describing data dependency and maintaining inter-object consistency is desirable. Existing consistency management models, such as the Smalltalk Model-View-Controller (MVC) framework[1], Dannenburg's ItemList[4], Garnet's unidirectional constraints[5], and ALV's bidirectional constraints[6], provide direct support for only some of the above capabilities.

This paper describes a new software architecture for consistency management: Change Propagation and Response Graphs (CPRGs). *Components* encapsulate data and behaviour, for example a dialog box component encapsulates a dialog box's size, location and manipulation functions. *Relationships* connect components, for example, a dialog box component to the edit field and button controls it contains. Relationships are also used to propagate descriptions of changes that components undergo to related components. A component interprets these *change descriptions*, updating its state to maintain inter-component consistency. Components can also store change descriptions, which allows many additional conistency management techniques to be supported in a homogeneous way. These include multiple view consistency, undo/redo and versioning facilities, lazy and incremental consistency management, and inconsistency management. The software architecture for CPRGs encapsulates these facilities, so programmers do not have to use ad-hoc implementation techniques to use them.

The paper commences with a description of common consistency management requirements. The basic CPRG architecture is then introduced, followed by a discussion of CPRG applications that illustrate the flexibility and homogeneity of the approach. An implemention of the CPRG model in an object-oriented programming language is described followed by a description of experience using this framework. The CPRG approach is then compared and contrasted to existing consistency management approaches.

2. COMMON CONSISTENCY MANAGEMENT REQUIREMENTS

Figure 1 shows a screen dump from MViewsDP, an interface builder for constructing dialog boxes[7]. This will be used as an example to illustrate consistency management needs common to a wide variety of applications. MViewsDP provides a graphical represention (view) of a dialog (window 'dialog1-dialog'), which is interactively edited to add and layout dialog box controls. One or more textual views permit specification of additional dialog functionality, such as constraints on data types entered, together with an interface to the dialog box to be used by programs (window 'dialog1-Dialog Predicate'). The third type of view shows a running prototype of the dialog specification which may be interacted with to test the specification.

When an edit field control in the graphical dialog specification view is dragged, its caption must be moved the same amount. This illustrates simple attribute dependency between the X and Y co-ordinates of the objects which describe the edit field and its caption. Similar attribute dependency occurs when the dialog box is resized: the dialog's controls are automatically scaled and/or moved to keep them inside the dialog's border and preserve their relative layout. More complex attribute dependency occurs between the dialog's interface (a description of the variables passed to and from the dialog when it is used by a program) and the dialog's controls. For example, when a new edit field is added to the dialog, the dialog's interface must be extended to add variables which hold the initial and final edit field values. Referential integrity between objects also needs to be supported. For example, when an edit box is deleted its caption must also be deleted, and the object describing the dialog box updated so that it no longer references these deleted control objects.

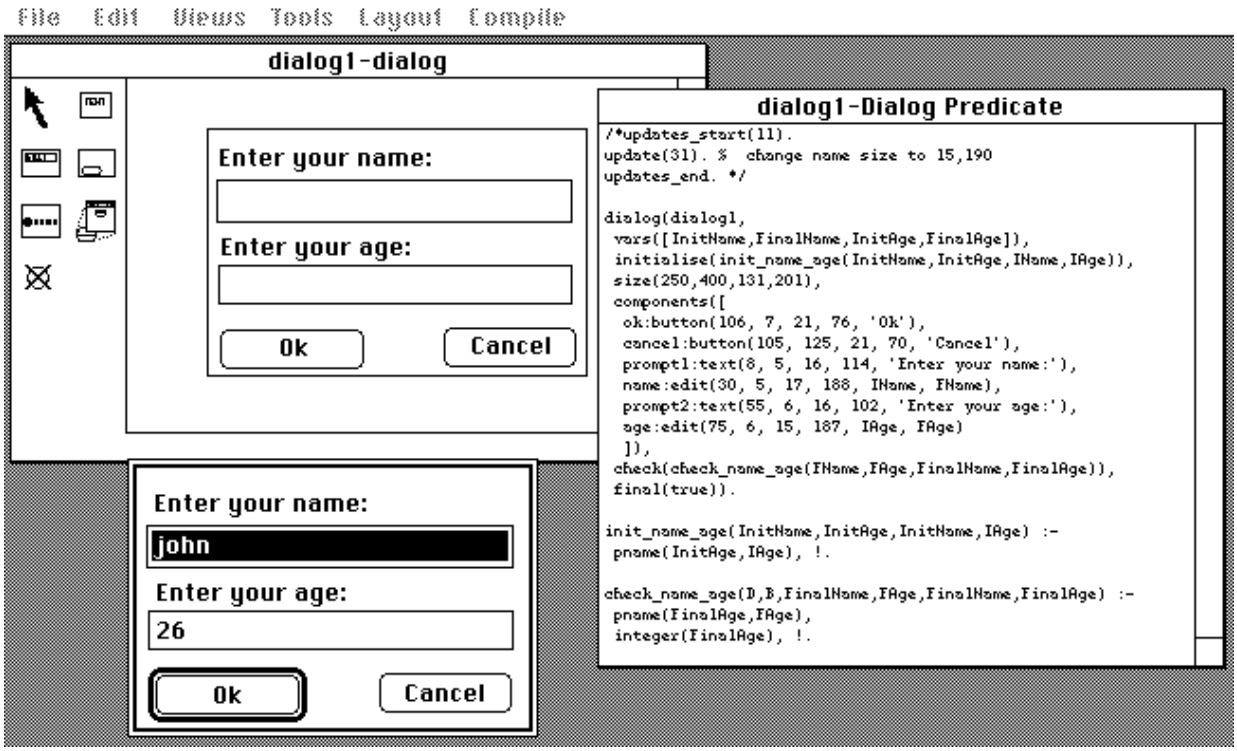

figure 1. A screen dump from MViewsDP.

As MViewsDP supports multiple views of a dialog box specification that share information, changes to one view must be automatically reflected in other views. This is another form of attribute dependency, although it is complicated by MViewsDP having to keep not only multiple views consistent, but different kinds of views (for example, graphical and textual representations of the same object). Some changes to each type of view can potentially affect the other view(s) in ways which cannot be automatically applied by MViewsDP. For example, adding a new edit field control in the graphical view should result in a new edit field description being inserted into the textual view. The control name and initial/final variable values of this new textual edit field are not specified in the graphical view, however, nor the position it should occupy in the textual view's interface description. Users need to be informed of such *partial* view updates, but may need to complete their implementation themselves.

Various inter-object constraints that are more complex than simple attribute dependency are required. For example, dialog contols may be constrained so that they cannot be dragged outside the confines of their bounding dialog box. Another example is after adding, updating or deleting a dialog box control, the dialog's interface must be checked to ensure duplicate variable and control names have not been used.

Other useful facilities that relate to consistency management include: an undo/redo facility, which allows users to undo or redo a sequence of view edits; version control, so users can revert to older versions of dialog specifications or merge two alternative specifications; and multi-user collaborative dialog specification and use. Lazy and incremental consistency management is also often required to help produce an efficient application, where reponse time to user edits is minimised. All of these latter facilities require descriptions of view edits to be stored, for later manipulation by the environment and/or its users. A general software architecture is required to support this wide range of consistency management techniques, in order to avoid application developers having to program them all in an ad-hoc fashion.

3. THE BASIC CPRG ARCHITECTURE

CPRGs support all of the above consistency management requirements by building upon a simple, kernel mechanism: discrete change description propagation along inter-object relationships, and response to and storage of these change descriptions.

Figure 2 illustrates how a dialog box specification from MViewsDP is represented using a CPRG. Each part of the dialog box specification is represented as a graph *component* (object), and these components are linked via *relationships*. For example the `dialog` component is related to its constituent parts via the `parts` relationship. Relationships are also components (objects) and thus can themselves be linked via further relationships. All graph components may have attributes describing their state. For example, the `dialog` component has attributes `name` and `interface`.

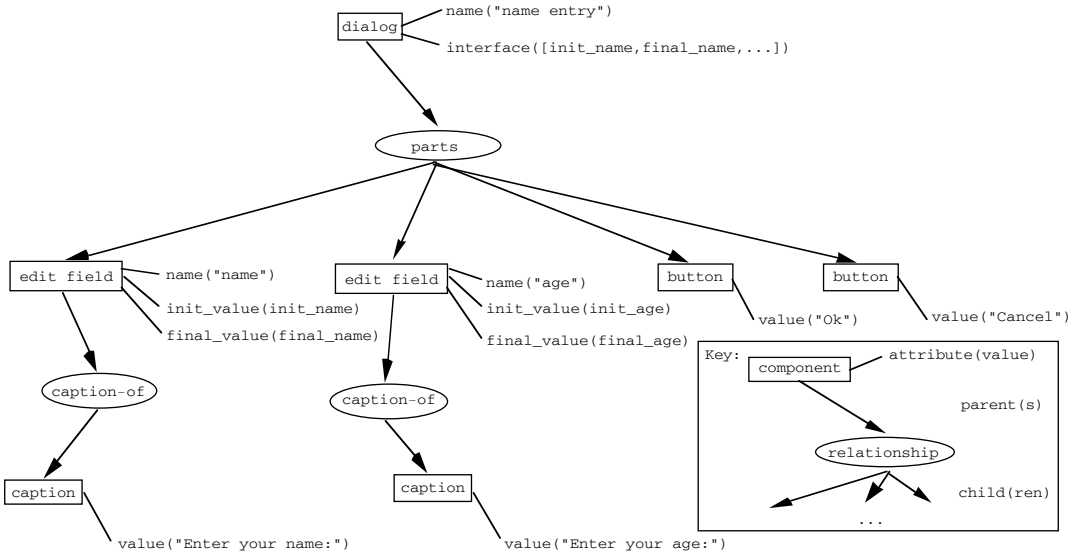

Figure 2. Representing a dialog box design using a CPRG.

CPRG graph components are modified by graph *operations*, which include adding and deleting components, establishing and dissolving relationships, and updating attributes. Referential integrity is enforced in CPRGs. When a component is deleted, any relationships it participates in are automatically dissolved.

Whenever an operation is applied to a graph component, a *change description* is generated. Change descriptions are of the form `<UpdatedComponent, KindOfUpdate,Value1, Value2, ..., Valuen>`, where:

- `UpdatedComponent` is the component which was updated by the operation;
- `KindOfUpdate` is the kind of operation which was applied to the component e.g. `update_attribute, delete_component, establish_rel,` etc.
- `Value1, Value2, ..., Valuen` are values associated with the kind of update e.g. `OldValue` and `NewValue` for `update_attribute`, `Parent` and `Child` for `establish_rel`, etc.

Following generation, change descriptions are propagated to all of the relationships the generating component participates in. These relationships then interpret the change description and take one of the following actions: 1) apply further operations to themselves or related components to keep the CPRG state consistent with the modified component; 2) pass on the change description to other related components; or 3) ignore the change to the component's state.

This change propagation mechanism supports simple attribute dependency by having a component update its own attribute values in response to change descriptions generated by attribute updates of a related component. Figure 3 shows an example of this simple attribute dependency supported by the basic CPRG model.

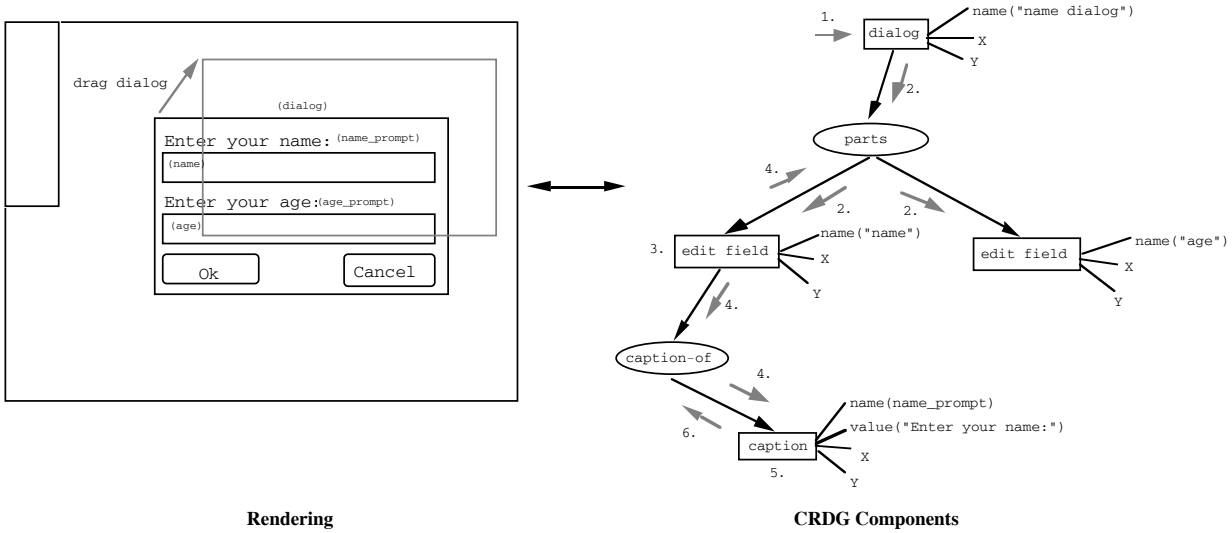

Figure 3. Maintaining attribute consistency between related components.

In this example, the dialog box border has been interactively dragged to a new location in the graphical specification view. This necessitates the dialog's CPRG components being updated so that they are moved the same amount as the dialog they belong to: 1) Dialog box `dialog` is interactively dragged, resulting in its CPRG component's X and Y co-ordinates in the editor window being updated; 2) `dialog` propagates change descriptions describing its state changes to its `parts` relationship, which forwards these onto the `name` and `age` edit field controls; 3) `name` and `age` respond to the change descriptions by applying operations to update their own editor window co-ordinates, to be consistent with that of `dialog's` (i.e. `name` and `age` are constrained to follow `dialog's` positional changes); 4) `name` propagates the change descriptions its operations produced to its relationships `parts` and `caption-of`. `parts` propagates the descripions to `dialog`, which ignores the changes in `name's` state, as it is not constrained to follow its controls' positional changes. `caption-of` propagates the change descriptions to the caption control `name_caption`; 5) `name_caption` responds to the change description from `name` and updates its own co-ordinates, so that it is dragged the same amount as `name` (i.e. `name_caption` is constrained to match `name`'s positional changes); 6) `name_caption` propagates change descriptions to `caption-of`, which forwards these to `name` which ignores these updates to `name_caption`, and propagation stops. The graphical renderings of the changed CPRG components are redrawn, and textual views annotated, to reflect their changed states.

Inter-component constraints are managed in a similar manner to attribute dependency. For example, in figure 3 the edit field control `name` may be dragged by interactive editing. The border of `name` should not be allowed to overlap the border of its owning dialog component `dialog` (as the edit field would not now fit inside the dialog box). This can be achieved by having the `dialog` component check any change descriptions generated by `name`. If `dialog` detects that `name's` border now overlaps its own, it can either abort the editing operation (undoing the invalid update), or have `name` rerender itself in a different colour (e.g. red), indicating an error.

4. RELATIONSHIP SPECIALISATIONS

CPRG relationships can encapsulate complex responses to different kinds of change descriptions, independent of the components they relate. Some such behaviours are quite general and can be used in a wide variety of applications for various kinds of related components. This suggests the need for a variety of

generic relationship types, such as aggregation (part-of) relationships, inter-component attribute dependency relationships, and multiple view (view-of) relationships.

Other behaviour encapsulated by relationships can be very specific to certain kinds of related components. For example, the parts relationship from figure 3 could implement the kinds of attribute dependency and constraints between a dialog component and its controls described in the previous section. When `parts` receives X and Y attribute change descriptions from `dialog`, it can apply operations to `dialog`'s controls to constrain them to follow `dialog`'s positional changes. Similarly, when `parts` receives X, Y, depth or width attribute change descriptions from any control components, it can check their borders still lie within that of `dialog`.

This kind of relationship between a graphical user interface object and related objects could be useful in many different applications, not just MViewsDP. Thus it would be useful to abstract out such behaviour into generic, reusable relationships. This section illustrates how different kinds of generic relationships can be defined, and then tailored for use in different applications, by extending the kernel CPRG model.

4.1. Aggregation Relationships

Software system components are often composed (or *aggregated*) from other components. For example, a radio group control is composed of individual radio button controls, which are in turn composed of a caption and button control. Often there is a transitive dependency in such situations between a component and the aggregates of another component. For example, if the caption of one radio button is modified, the enclosing box around the radio group may need to be enlarged to continue to enclose the new caption. Thus the radio group as a whole is dependent on the aggregates of each of its radio buttons. Figure 4 shows an example of this situation. The `parts`, `radio-parts`, `caption-of` and `button-of` relationships are specified as instances of the *aggregation* (part-of) relationship type. Aggregate relationships automatically forward any change descriptions generated by children onto the parent of the relationship and have the parent broadcast this update to its relationships. This makes it appear to the parent's relationships that the parent component itself has been updated.

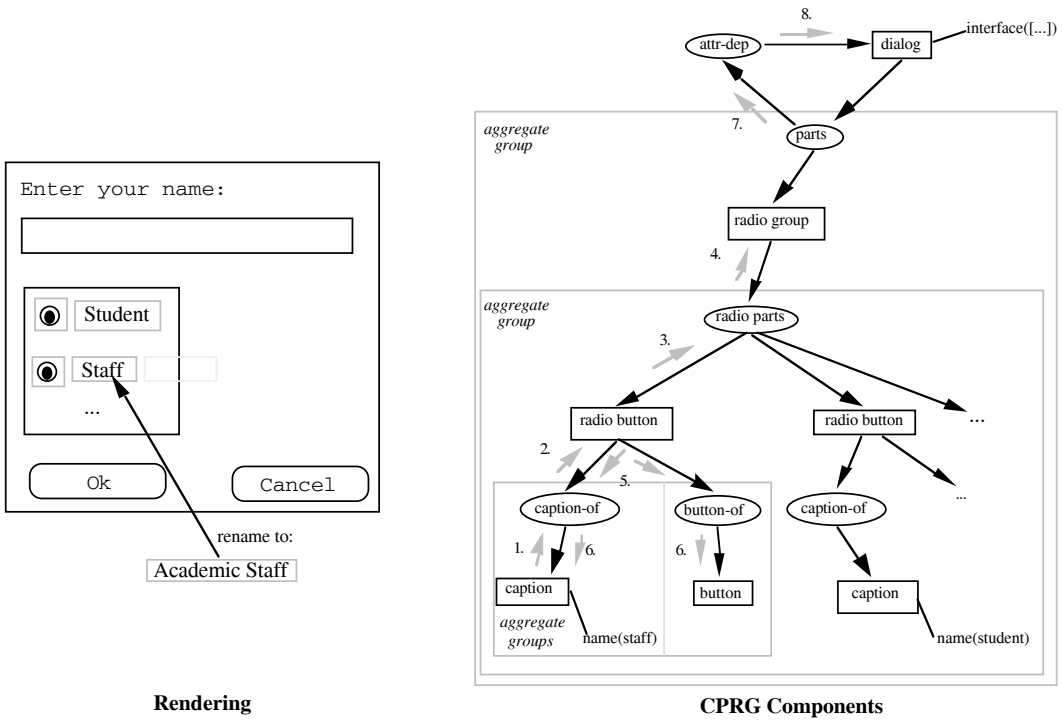

FIgure 4. Aggregates and attribute recalculation.

In this example: 1) the name of the caption control is modified to 'Academic Staff', generating a change description; 2) the caption-of relationship forwards this to the parent of the aggregation relationship, radio button; 3) radio button propagates the update to all of its relationships; 4) the radio parts relationship forwards this to radio group, which then forwards the change description to all of its relationhips. radio group's response to the change will determine that one of its aggregates has been changed and that its enclosing border needs to be expanded to continue to enclose all of its aggregates.

In the reverse direction, aggregation relationships can be used to allow a parent component to apply the same operation to all of its aggregated components. An example in figure 4 is where: 5) the Staff radio button is deleted; 6) `caption-of` and `button-of` respond to this change description by deleting the caption control and itself. As a more general example, `radio group` can get `radio parts` to apply an arbitrary operation to all of its aggregate controls, for example "draw" or "hide".

Often applications will specialise the generic CPRG aggregation relationship to specify additional application-specific functionality. For example, the `parts` relationship in MViewsDP specialises the aggregation relationship to additionally enforce constraints on the names and enclosing borders of dialog box controls.

4.2. Attribute Dependency Relationships

Many systems support some form of attribute recalculation or attribute constraints i.e. define an attribute's value to be an expression over other attribute values, and recalculate the attribute when any of the other attribute values change[8, 9, 10]. CPRGs allow efficient attribute recalculation to be implemented via "attribute dependency" relationships. Applications tailor attribute dependency relationships for use by supplying the dependent attribute's name and the expression to use when recalculating the attribute's value. For example, in figure 4, an attribute dependency relationship is defined to recalculate the value of a dialog box's `interface` when change descriptions are received from dialog controls.

In this example: 7) a control of `dialog` has been updated (eg renamed), resulting in a change description being propagated to `attr-dep` via the `parts` aggregation relationship; 8) `attr-dep` recalculates the `interface` list attribute of `dialog` (either incrementally or in full), keeping this attribute consistent with the changed state of `dialog` and its controls. When this attribute is recalculated, it also generates a change description, and this will trigger the recalculation of any attributes dependent on the value of `interface`, or checking of any constraints on the value of `interface`.

Attribute dependency relationships can be used to implement flexible bi-directional constraints, by supplying additional responses to perform the reverse of the above. For example, if a control name in the `interface` list is changed, generating a change description, the `attr-dep` relationship responds to this by determining the appropriate control to rename. This is straightforward, as the change description holds the before/after name of the updated `interface` list value. Similarly, if a value was added to or deleted from the `interface` list, `attr-dep` could respond to this by adding or deleting the appropriate control.

The incremental attribute recalculation schemes of many systems support the full recalculation of attribute values when a value the attribute depends on is changed[9, 10, 11]. CPRGs additionally allow complex attribute values to be incrementally recalculated, using change descriptions describing the changes to values they depend on. For example, if the `name` edit field's initial value was changed from `init_name` to `init_val`, most other attribute recalculation schemes would have to fully recalculate the value of `interface` for `dialog`, by re-evaluating a function over the controls related to `dialog` by `parts`. As CPRGs document the exact change a component has undergone via change descriptions, the `interface` list can be more efficently updated by having the old `init_name` value removed and a new `init_val` item inserted, leaving the rest of the `interface` list unchanged. We have found this technique to be very efficient where complex attribute data values (such as lists or other collection data structures) often have single items added, removed or updated. It is not so effective when large numbers of changes are made to the data structure, but in this case a conventional full recalculation can be done (the change descriptions simply indicate the whole attribute value must be recalculated).

4.3. View-of Relationships

Many systems support a notion of multiple views of an application-level component[8, 1, 12, 13]. These "view components" can often be interactively edited, which results in the application-level component being updated. All other views are then updated appropriately to keep them consistent with the updated application-level component.

View consistency can be achieved with CPRGs via a generic view-of relationship, which relates application-level components to one or more view-level components, as shown in figure 5. The default form of view-of relationships maintain multiple view consistency by translating changes in application-level component attributes into changes on their view component attributes, and vice versa. Applications tailor the view-of relationship for use by defining mappings between application-level and view-level component

attributes. They can also be specialised to define the effects on a view component when application-level components are added, deleted or have relationships established to other components, and vice versa. MViewsDP defines a range of view-of relationship specialisations, one for each kind of dialog component, control component and relationship, and for each kind of view component representation. These not only keep component attributes consistent but automatically expand and hide appropriate view components when application-level dialog controls are added or deleted.

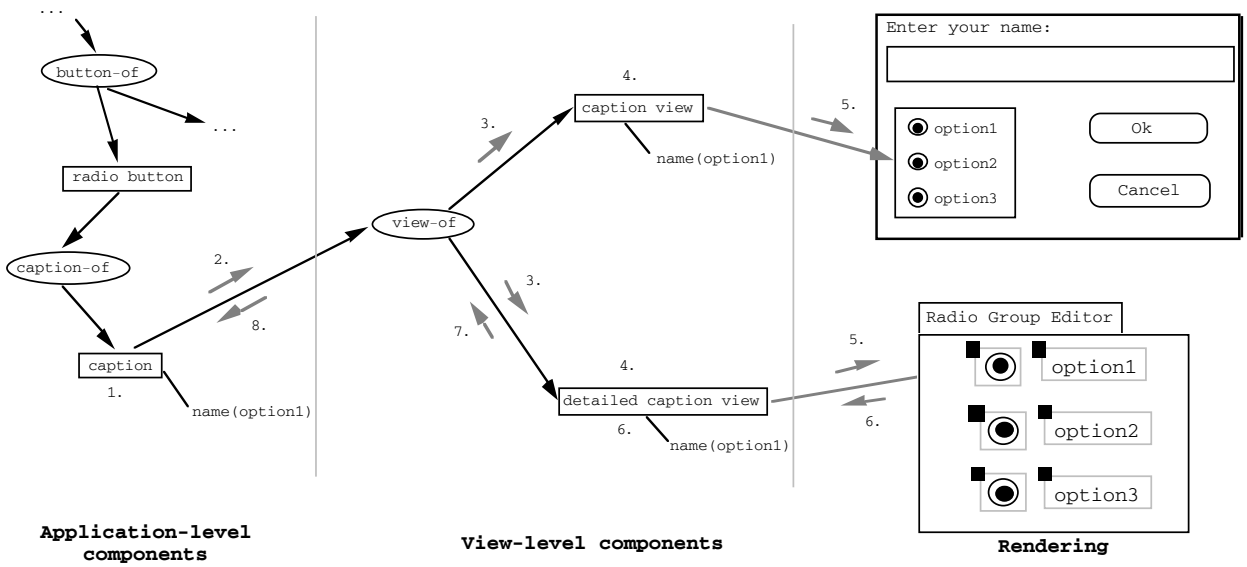

Figure 5. Example of multiple views using CPRGs.

Figure 5 shows how this multiple view consistency mechanism works for two graphical views of a dialog component (one a high-level dialog overview, the other a detailed view of a dialog radio box and its controls): 1) an application-level radio button caption `option1` is updated (e.g. renamed); 2) `option1` generates change description(s) which are propagated to all relationships of `option1`, including a `view-of` relationship; 3) `view-of` translates the application-level component operation(s) into view-level component operation(s), updating view component names, by responding to the change description(s); 4) the view-level components are updated by the operations; 5) view-level components are rerendered to reflect their updated states.

This process also works in reverse, where a view-level component is interactively updated: 6) user interactively edits detailed view, renaming the view component, resulting in a view-level component update operation; 7) change description(s) are propagated to `view-of` relationship; 8) `view-of` relationship translates view-level operation(s) into application-level operation(s) by responding to the view component change description(s).

4.4. Partial View Consistency

As identified in section 2, maintaining view consistency is not always straightforward, as an environment or user interface can not always automatically keep different representations consistent, requiring partial view consistency techniques. An example is where a dialog component has two views, one graphical and one textual, which share overlapping information and thus must be kept consistent. Some changes made at the graphical view level can not be completely translated by the environment into textual view updates, and vice versa.

CPRGs document modifications to application-level component as change descriptions. These descriptions can be displayed in a readable format in either a dialog or directly inserted into textual views to indicate related changes have been made in another view. Figure 6 shows a change description inserted into a textual dialog specification's text. This indicates an update has been made on the dialog's `name` control in a graphical view and that the two views are now only partially consistent with one another. Where a directly translatable update has been made in another view, as is the case in figure 6, users can interactively select these change description representations and request the environment make the appropriate view update(s) to restore consistency. Where automatic translation is not possible, these readable change descriptions act as documentation informing users the view must be manually changed to restore consistency. Change

descriptions can also be displayed in readable form in dialog boxes, or as textual annotations to graphical view components.

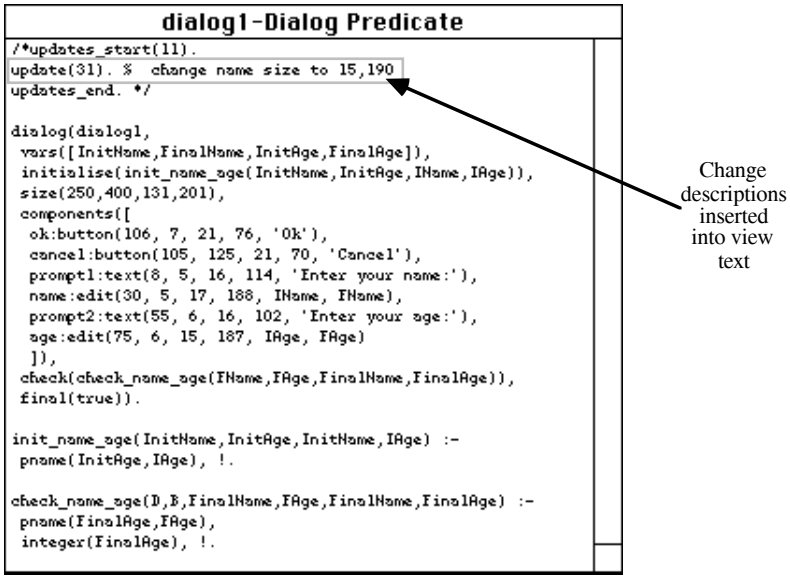

Figure 6. Textual view consistency via change description display.

4.5. Other Generic Relationships

Applications can specialise the generic relationships described above to perform additional consistency management tasks. Many of these specialisations are useful generic relationships in their own right. For example, many graphic constraints can be packaged into generic relationships, such as constraining one component's border to always be inside another's border, or constraining one component to always be aligned with another component. MViewsDP uses such generic relationships for the graphical specification view. It also uses a "unique values" constraint for aggregates, which constrains the value of a key attribute of all aggregated components to be unique for each component, to ensure unique dialog control names.

4.6. Combining Relationship Behaviour

Relationships exhibiting multiple types of behaviour are often useful in an application. For example, the `parts` relationship in MViewsDP encapsulates aggregation, graphical constraints, and the unique value constraint. These different relationship behaviours can be combined in one of three ways: 1) a generic relationship of each kind can be established between the components; 2) multiple inheritance (if supported by the implementation language) could be used to combine the responses of each relationhip, unifying their behaviour; or 3) a new relationship can be defined, encapsulating all of the consistency management behaviour as its change description response. The first approach is easiest, although it has the overhead of maintaining several relationships between the same components. The third is potentially the most efficient, but the most work to implement, and makes the resulting relationship less reusable. The second approach is useful when the implementation language for CPRGs supports multiple inheritance. Relationships can also be "chained together" by having a relationship relating a component to another relationship. Complex change description transformations can thus be be propagated along a series of connected (possibly generic) relationships between two components. In MViewsDP, the parts relationship multiply inherits from the generic CPRG aggregation relationship, a unique values constraint relationship, and defines extra application-specific behaviour to constrain dialog control borders.

5. CHANGE DESCRIPTION STORAGE

Change descriptions are treated as first-class objects in the CPRG model, and thus they can be stored and manipulated, and programmers do not have to write any ad-hoc mechanisms for storing descriptions of changes for undo/redo mechanisms or version control facilities. This is in contrast to most other systems, such as the MVC[1], active data structures[14], and most constraint systems, such as ALV and Garnet[6, 5]. These systems have to translate object and attribute state changes into storable forms in programmer-defined ways, in order to implement many of the consistency management techniques described in this section.

5.1. Undo/Redo and Editing Transaction Facilities

For user interfaces and editors, stored change descriptions can be used to support both generic undo/redo and editing transaction facilities. These are fully compatible with the other consistency management techniques, as all are driven by change description propagation.

In the example in figure 7: 1) an interactive update to a view component results in change descriptions being generated; 2) these change descriptions are propagated to a `view_editor` component via a `view-component` relationship; 3) `view_editor` stores the generated change descriptions in an `undo_history` list attribute; 4) the user requests the last editing operation for the view be undone (e.g. by selecting an "Undo" menu item); 5) to achieve this the `view_editor` sends back the change description(s) associated with the last view edit to their generating component(s) for reversal; 6) the component(s) undo the editing change by applying operations to themselves which reverse the updates specified by the change descriptions they initially generated; 7) new change descriptions are propagated which reverse the effects of e.g. attribute recalculation, application-level component updates, lazy consistency management, etc. Note that the view_editor only stores the initial change descriptions generated in response to view editing operations. Any additional change descriptions generated by propagating these inital change descriptions, or undoing or redoing stored change descriptions, are thrown away, as only the initiating change description is required to undo the edit.

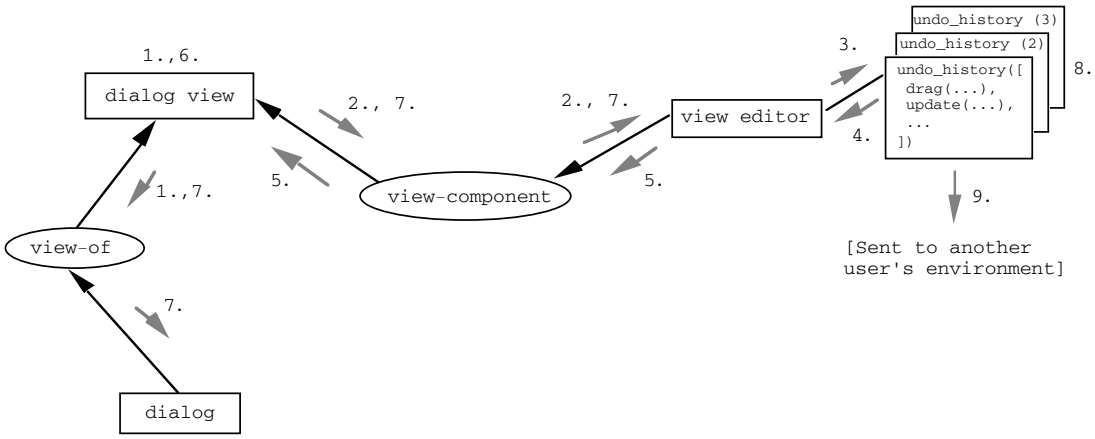

Figure 7. An undo/redo facility, component versioning and cooperative work.

An editing transaction may be aborted when some constraint is violated by an editing operation, so that any component updates made during the edit are reversed. This is achieved by a component requesting the view editor component to reverse all of the change descriptions stored for the current editing operation. These are reversed by sending them back to generating components, as for undo. View components are always rerendered after update, maintaining the correct rendering for the view.

5.2. Version Control and Cooperative Work Facilities

Version control is supported in a similar manner to undo/redo histories, by storing change descriptions in *version records*, which group changes applied to the same version of a component. These grouped change descriptions are undone or reapplied to regenerate previous or subsequent versions of a component in the same manner as view undo/redos. In the example in figure 7: 8) the view may have several "undo history" lists, which in fact describe the changes made to the view to generate different versions of the view's components. Computer-Supported Cooperative Work (CSCW) facilities are also built using this model: 9) change descriptions broadcast between different users' environments provide a change communication medium. Both synchronous and asynchronous editing can be supported by using this change description broadcasting approach[15].

5.3. Inconsistency Management

Many consistency management systems assume the system is always consistent i.e. constraints are "hard" and are always enforced[6, 9, 5]. This includes systems which support lazy consistency management (see 5.5), as at some stage all component states must be reconciled and made consistent[16]. CPRGs allow change descriptions to be stored in attribute lists to document inconsistencies between components, without

necessarily resolving these inconsistencies. This is useful where there doesn't exist any mechanism for automatically maintaining consistency, or where users of the environment wish to be informed of certain inconsistencies and resolve these themselves.

Examples in MViewsDP include the textual view consistency mechanism, where a textual view can be inconsistent with the application-level components. Descriptions of the application component changes are "stored" in the view's text, however, documenting the inconsistency. Another example is where the interface list attribute can contain same-named variables. Rather than enforce a unique names constraint, the environment can store a change description documenting this error and allow the user to correct the problem. This is a similar approach to the "error attributes" of the Cornel Program Synthesizer[10]. It is more flexible, however, as the stored change descriptions are generated by the environment, and can be used, for example, to reverse the affect of an inconsistent operation at a later stage. Viewpoint inconsistency[17] uses logic predicates to record inconsistencies, but it is unclear on how users view or interact with these inconsistent aspects.

5.4. Change Description Composition

Storage of change descriptions also permits a powerful abstraction mechanism to be implemented: change description composition. Change descriptions are seldom generated in isolation: it is often desirable to group sequences of discrete change descriptions and to capture extra information about the group of related graph updates. *Macro* change descriptions are composed of a sequence of discrete change descriptions or other macro change descriptions. A macro change description is propagated to a related component and the related component can respond to this single description, or it can decompose the macro description into its individual change descriptions and respond to (some or all of) these. This is generally more efficient than propagating and storing each individual change description as it is generated. The individual change descriptions can sometimes be thrown away when storing the macro change desription, as undo/redo often only need to know the composite component operation to apply to undo/redo the affects of the whole group of operations. This greatly reduces change description storage requirements. Users viewing the macro change description in a dialog box or textual view usually prefer to see only the macro change description rather than all of its constituent change descriptions.

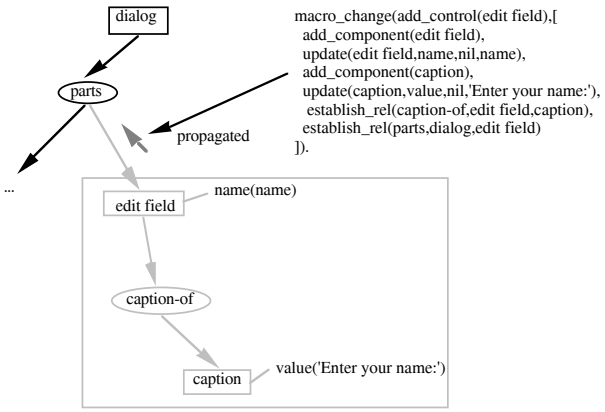

Figure 8. Change description composition.

For example, figure 8 shows an example of change description composition when adding a new edit field control to a dialog box specification. It is more useful to store and present to users the macro change description `add_control`, rather than all of its individual change descriptions. In addition, it is more efficient when undoing such an operation to undo the affects of `add_control` by simply deleting the new edit field component (which will result in the aggregated caption and relationships also being deleted). The default way of undoing macro change description affects is to undo the affects of each individual change description in reverse order, achieving exactly the same result but much less efficiently.

5.5. Lazy Consistency Management

Lazy consistency management is the ability to make component states consistent after several operations have been applied in sequence[17, 16]. To support this in the CPRG model, change descriptions are either stored in list attributes of components before they are propagated to relationships, or are stored by relationships before they are propagated onto related components. At some later stage the sequence of

stored change descriptions is actioned as a group: propagated onto relationhips or related components; translated into operations on a component; or manipulated in some way. This storage and subsequent actioning of change descriptions supports various forms of lazy consistency management, including lazy view consistency (view-of relationships do not action stored change descriptions until a view is displayed) and lazy attribute recalculation and constraint (attribute values are not recalculated until another component requires the attribute value).

6. AN OBJECT-ORIENTED FRAMEWORK FOR CPRGS

We have implemented an object-oriented class framework implementing CPRGs. Classes from this framework are specialised to reuse the CPRG model and avoid programmers having to implement most aspects of the consistency management techniques described previously. Fig 9 shows the basic classes in this framework. Labelled lines represent whole-part relationships between classes of objects. Names within class icons represent features (attributes and methods) supported by each class of component.

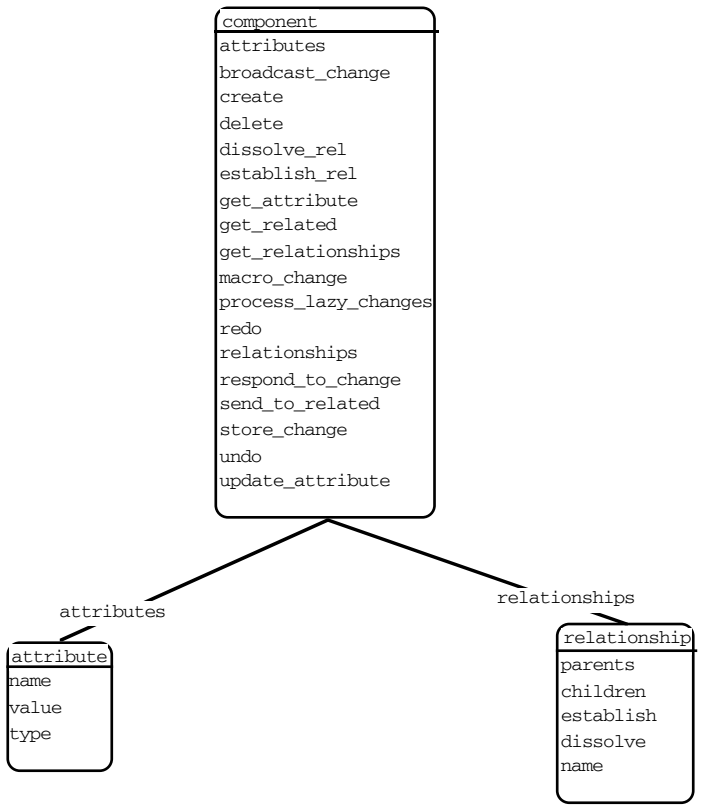

Figure 9 The class structure modelling CPRG components.

CPRG components have a list of named, typed attributes, and a list of relationships the component participates in. Relationships are themselves components, as can be seen from Fig. 10, which shows the inheritance hierarchy (bold arrowed lines) between classes in the CPRG framework.

Specialisations of the `relationship` class model inter-component relationships of different arities. Part-of, view-of and attribute dependency abstractions model aggregation, partial view, and attribute dependency relationships between components. "Active" data structures, similar to those of[14], are also supported, and include lists, hash tables and B-trees. These structures inform dependent components of changes to their state (inserts, removals, updates, and creation and deletion) via change descriptions. They also provide specialised one-to-many relationships which either index children components for efficient retrieval, or maintain an ordered sequence of children components (the standard one-to-many relationship orders the children in an arbitrary way). Programmers can often reuse these generic relationships and active data structures without having to specialise them.

Applications specialise framework classes to implement application-specific component structures. These new structures use the CPRG model for consistency management. For example, MViewsDP specialises the

component class to describe `dialog` and `edit_field` controls and specialises the `part_of` relationship class to describe the dialog `parts` relationship.

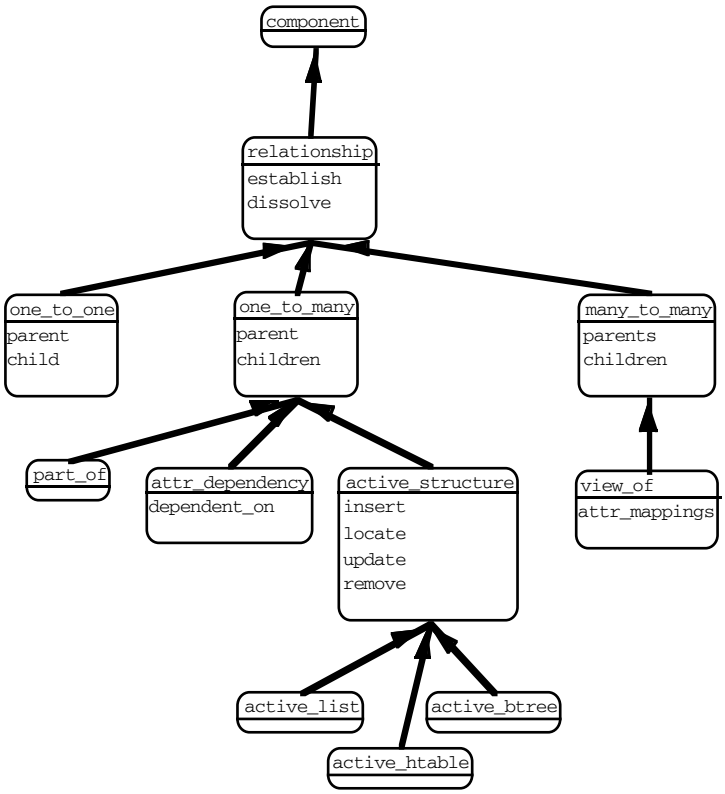

Figure 10 The CPRG framework.

Component operations are implemented as methods. An operation is applied to a component object by calling the relevant method with appropriate arguments. Table 1 briefly describes the basic CPRG methods and the change descriptions they generate (if any). Specialisations of CPRG components can provide additional operations composed out of these fundamental operations (implemented as new methods), and can define new specialised change descriptions.

Components implement a "change description broadcaster" method, `broadcast_change(ChangeDescription)`, which informs all relationships connected to the component of the update to the component. Components also implement a "change description response" method, of the form `respond_to_change(ChangeDescription, SendingComponent)`, which determines the component's response to change descriptions it receives. Relationships can process change descriptions and apply operations to their parent(s) or child(ren) components, providing a powerful mechanism for abstracting out common relationship functionality into reusable relationship component classes.

Operation (Method Call)	Change Description Generated	Component State Change/Result
Value = Comp.get_attribute(Attribute)	-	Get value of Attribute for Comp
Comp.update_attribute(Attribute,New)	update(Comp,Attribute,Old,New)	Update Attribute of Comp changing Old value to New
Comp.create	add_component(Comp)	Create a new component Comp
Comp.delete	delete_component(Comp)	Delete component Comp
Comp.establish_rel(RelName,Parent,Child,Rel) or Rel.establish(Parent,Child)	establish(Comp,RelName,Parent,Child) or establish_link(Comp,RelName,Child)	Establish a relationship of type RelName between components Parent and Child (may be simple link relationship, for efficiency)
Comp.dissolve_rel(RelName, Parent,Child,Rel) or Rel.dissolve(Parent,Child)	dissolve(Comp,RelName,Parent,Child) or dissolve_link(Comp,RelName,Child)	Dissolve the relationship of type RelName between components Parent and Child (may be simple link relationship, for efficiency)
Value = Comp.get_related_comps(RelName)	-	Get all components related to Comp by relationship(s) of type RelName
Value = Comp.get_relationships(RelName)	-	Get all relationships of type RelName that Comp participates in
Comp.send_to_related(RelName,Change)	Change	Send change description Change to all components related to Comp by relationships of type RelName
Comp.broadcast_change(Change)	Change	Propagate change description Change to relationship components Comp participates in
Comp.store_change(List,Change)	-	Store change description Change in list attribute List of component Comp
Comp.respond_to_change(Change,FromComp)	-	Respond to change description Change from component FromComp
Comp.process_lazy_changes(List)	-	Process all change descriptions stored in list List
Comp.undo(Change)	-	Undo affect of operation which generated change description Change
Comp.redo(Change)	-	Redo affect of operation which generated change description Change
Comp.macro_change(Change,Operations)	Change(Comp,ListofChanges)	Compose all change descriptions generated by Operations into macro change description Change. Then propagate the group as a whole to the component's relationships.

Table 1. Basic component operations supplied by CPRG components.

Components always determine other components they are related to, and the relationships they participate in, by calling `get_related_comps(RelName)` or `get_relationships(RelName)`. This maintains referential integrity, as when the `delete` method of a component is called, all relationships connected to the component are informed of the impending deletion and are dissolved. Part-of relationships delete all their child components, and view-of relationships render their view components as "has deleted base", if the viewed component has been deleted. When a component object is deleted, it is added to a "lazy deletion" list after its relationships have been dissolved. This ensures that any lazy update record processing is performed before the object is actually garbage collected.

Cycles can sometimes occur during update propagation, and can be handled in one of two ways. The basic component operations, such as update attribute etc., can be restricted to occur only once with the same arguments on each component, for each distinct view editing operation. Thus if two updates of the same attribute with the same value are attempted on a component during an editing operation, only the first actually succeeds and generates an update record, thus breaking the propagation cycle. The second approach involves passing a list of previously applied operations during the cycle as an argument to each component's operation methods (or allowing this to be accessed globally). This provides propagation "path" information, which can be used by `update_from` methods to determine if a cycle has occured.

An editing cycle is always used with CPRGs to maintain graph consistency. A user initially manipulates a CPRG rendering or requests parsing of a textual view. This is translated into graph operations by the view editor or parser, updating view components. This update generates change descriptions which are propagated to view-of relationships, which then apply operations to update the application-level CPRG components. Further change descriptions are generated, which are propagated to all view-of relationships. All view components affected by the original view modification are then updated, maintaining multiple view consistency. Any change descriptions scheduled for lazy processing are then actioned, possibly generating further change descriptions. These are propagated to all affected components and may schedule further lazy processing. Once all lazy processing is complete, any deleted CPRG components are garbage collected, as the CPRG referential integrity mechanism has ensured no object references remain to them.

The automatic update of free-edited textual views in response to change descriptions is more complicated than most other systems, which either keep structure-edited textual views consistent, such as Dora[12], or use restricted, ad-hoc techniques, such as in FormsVBT[18]. Dora environments keep structure-edited graphical and textual views consistent, but do not support any partial view consistency techniques. FormsVBT keeps free-edited graphical and textual views consistent under change. However, it requires that the textual and graphical views represent exactly the same information and that textual view updates are prevented while the graphical views are updated, and vice-versa. MViewsDP is much more flexible, allowing either view to be

updated at any time and does not require a full token array of the text to be kept at all times, as does FormsVBT.

Using the CPRG architecture, environments can automatically apply some updates to textual views, i.e. update the view's text, rather than just insert a readable description of the change description in the view header. An environment implementer must specify a regular expression used to parse the view's text to find where to make an appropriate modification, and must specify how to unparse the change description into changes to the affected text component. This incremental unparsing process works well when the elements in the text to be updated are uniquely identifiable, or when the elements to update can be identified by context i.e. by the lexical tokens that surround them. For aspects not uniquely identifiable, the text component can be annotated with tokens which appear as comments in the language being processed but have meaning to the lexical analyser.

Graphical view component renderings are redrawn if the component they represent has generated any change descriptions. This rerendering is done at the end of the editing cycle for the graphical view, to ensure components are only rerendered once. Graphical view components can store and interpret the change descriptions they generated and use these to perform efficient, incremental view rerendering, similar to the approach used by ItemLists[4].

7. FRAMEWORK IMPLEMENTATION

We have implemented a prototype of the CPRG framework in Snart[7], which provides a set of C++-like object-oriented extensions to Prolog. Application-specific components are implemented by specialising Snart classes from the CPRG framework. Instances of CPRGs are represented as object instances of these new Snart classes. Component operations are implemented as Snart class methods and method calling "applies" an operation to a component object. CPRG component attributes are implemented as Snart object attributes, and list attributes are stored as Prolog lists, both for extra efficiency.

The Snart implementation of CPRGs allows simple link relationships to optionally be implemented as direct object references or lists of object references, instead of as discrete objects, for efficiency. These link relationships just describe connectivity between components and do not encapsulate any behaviour, unlike relationship components. To be consistent with the CPRG philosophy, this requires these reference-based relationships to be correctly established and dissolved between components. The `establish_rel` and `dissolve_rel` methods support both relationship object and reference relationship establishment and dissolution and ensure CPRG connectivity and object references are always consistent.

As all operations generate change descriptions, change description generation, propagation and response must be very efficient for acceptable performance of CPRG-based systems. Our Snart implementation represents change descriptions as Prolog terms. An operation calls the `broadcast_change` method with a term argument (representing a change description) of the form $\texttt{KindOfUpdate(Component,Value}_1\texttt{,...Value}_n\texttt{)}$. This term is constructed at compile-time and thus only needs variables it uses to be instantiated at run-time by the Prolog system. Change description generation is thus very efficient.

To increase the efficiency of change description propagation, our Snart prototype permits a subset of the relationships a component participates in to be "dependent" relationships i.e. relationships informed of component updates. When first established, these relationships indicate to the parent and/or child components they wish to be sent change descriptions. This eliminates unnecessary change description broadcasts to "disinterested", yet structurally related, components.

When broadcasting change descriptions, the `broadcast_change` method of a component calls the `respond_to_change` method for each of its dependent components. The Snart implementation uses a declarative style of logic programming which specifies a set of patterns a change description must match for the dependent component to respond to a change description. This pattern matching is carried out by Prolog's unification algorithm. CPRGs also use declarative methods to specify how change descriptions are converted into their stored forms, how operations are undone and redone, and how change descriptions are displayed in dialogs and used to automatically update textual view text and graphical view icons.

Software applications usually need to store data structures between program invocations. Two common methods of achieving this data persistency are to write object data to files or to a database (relational or object-oriented). Our Snart CPRG framework supports data structure persistency using a third method, object persistence. Snart objects are dynamically made persistent, and this provides automatic data structure

persistency for any CPRG application using a Snart persistent object store. Snart's object persistency mechanism also supports automatic object schema evolution so updated CPRG classes can use old object structures. We have found this object persistence mechanism to be far easier to use than file or database approaches as the latter require additional component-specific class methods to be implemented for object saving and reloading.

8. EXPERIENCE WITH CPRGS

8.1. MViews

MViews[19] is a framework for constructing integrated software development environments (ISDEs) incorporating the CPRG model. MViews uses components and relationships to represent program structure and semantics, and to represent multiple views of these application-level components as view components. It also has abstractions for defining textual and graphical renderings of these view components and for building view editors. The MViews framework provides Unidraw-like abstractions for building graphical editors, and for keeping graphical view components consistent when their application-level components change. MViews supports free-edited textual views with parsers defined to update application-level structures after view editing. Automatic updating of textual views is supported by defining regular expressions which specify the character sequences (i.e. lexical tokens) which need to be updated for certain change description patterns. Incremental parsing of the view identifies the tokens which need to be replaced to update the view's text. We have recently developed C-MViews, an extension to MViews which supports component and view versioning and merging support, and facilities for building collaboratve environments[15].

The MViews framework has been used to construct several multi-view editing environments, including MViewsDP. Here we briefly describe another, SPE, to illustrate the versatility of the CPRG approach. Other environments constructed using MViews include: MViewsER, a multi-view ER diagrammer[7]; EPE, a development environment for EXPRESS and EXPRESS-G[20]; HyperPascal, a visual Pascal-like language[21]; Cerno, a visualisation system for object-oriented programs[22]; and the Skin programing environment, for visual definition of dynamic user interface components[23].

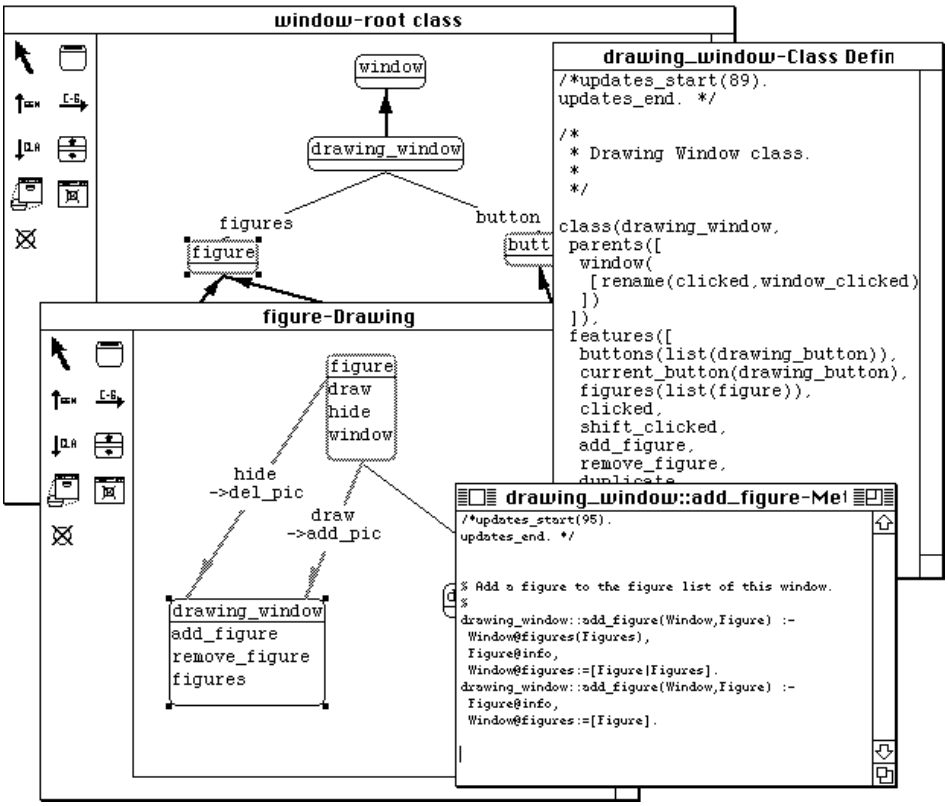

Figure 11. An example of SPE views during the development of a simple drawing program.

SPE (the Snart Programming Environment) is an ISDE for the Snart object-oriented language implemented using the MViews framework. It is described in detail in[22]. SPE supports multiple graphical and textual views of Snart program development. Figure 11 shows a screen dump from SPE during the development of a drawing editor program. Several windows are shown, each corresponding to a different view of the program. One graphical view shows an analysis-level diagram representing important generalisation and aggregation structures. The second shows a design-level diagram describing method calling protocols when rendering figures in a drawing window. One textual view shows a detailed class interface for the `drawing_window` class while another shows a method implementation.

SPE uses the CPRG model to represent program components and multiple views of these components, to record a detailed update history for classes, and to implement an undo/redo mechanism for view editors. Multiple textual and graphical views and component attributes are kept consistent via change description propagation, and both view updates and attribute recalculation use incremental change description processing for efficiency.

We have recently developed an extension to SPE, C-SPE, which provides collaborative support for object-oriented software production. This allows groups of developers to cooperate on software development using multiple views, facilitated by change description broadcasting between different developers' environments[15].

8.2. Performance

The main overheads of the CPRG model are attribute and relationship representation, and change description generation, propagation, response and storage. Using object attributes to implement component attributes, and optionally implementing relationships as object references, minimises the cost of the first two overheads. Change description implementation as Prolog terms is very efficient with little overhead for generating change descriptions. Propagation of change descriptions is currently restricted to only those relationships designated as dependents, a list of which are cached in a `dependents` list attribute (which may be recomputed lazily for efficiency). Declarative change description processing by components is compiled by Snart to take advantage of first-term indexing which Prolog compilers can optimise.

Snart currently runs under LPA MacProlog on Apple Macintosh computers. The Snart language itself is not currently optimised but LPA Prolog code can be optimised to improve performance. We have used CPRG applications on PowerBook 180, Quadra 800 and PowerMac computers, and application performance has proved very acceptable, even for large applications with thousands of CPRG component objects (such as SPE and EPE).

8.3. Portability

A prototype of the CPRG model has also been implemented using THINK C, a variant of C++, on a Macinosh computer. This used heap allocated memory rather than objects to implement efficient lists and attribute storage, similar to that employed by our Snart framework (as Prolog lists and terms). Change descriptions are constructed by defining C macros which allocate stack memory to generate change descriptions and references to these change descriptions are passed to propagate them. Heap-allocated memory versions of these change descriptions are only created if the change description is stored in a list attribute. Change description processing uses C switch statements, which are not as abstract as declarative Prolog predicates. The performance of this CPRG prototype is far superior to the Prolog version, although the functionality is not as well developed.

8.4. Complexity and Scalability

The CPRG framework is relatively simple and has proved quite easy to reuse, both by the developers of the framework and other researchers. The framework provides a set of low-level consistency management abstractions, however, and is primarily intended for implementing higher-level techniques. Many constraint systems and attribute grammar systems, for example, provide more abstract languages for specifying attribute dependency and constraint schemes. The CPRG model has performed well for systems such as SPE and EPE, where there are a large number of inter-related components. We are currently developing forms of MViews and SPE which support collaborative software development, including change description broadcasting between different users' environments. It is too early to determine whether or not the CPRG model will scale up for these applications, but initial results are promising.

9. COMPARISION TO EXISTING SYSTEMS

We have found CPRGs provide a very flexible approach to implementing a wide range of consistency management techniques. Several attempts have been made to provide a generalised model or framework to address some of the consistency management requirements outlined in Section 2. The challenge, however, is to meet *all* of these needs in a flexible, yet homogeneous fashion.

The main advantage of the CPRG approach is its homogeneous nature. Unlike other systems, CPRGs always generate discrete change descriptions when data objects are modified, and these change descriptions are always able to be propagated, responded to, and stored. This allows CPRGs to support a very wide range of flexible consistency management techniques all based upon the same kernel idea. Programmers do not have to invent ways of generating descriptions to store for undo/redo or versioning, as in most constraint-based systems, such as Garnet[5] and Rendezvous[6]. They also do not have to invent ways of keeping multiple textual and graphical views partially consistent. Partial consistency is not supported by most other multi-view editing systems, such as Dora[12], LOGGIE[8], Unidraw[3], and FormsVBT[18]. The graph-based structure of CPRGs, together with its active data structures, means they can be used to describe data structures more readily than list-based consistency models, such as the ItemList structure[4] and Object Dependency Graphs[16].

CPRG relationhips are more flexible for consistency management than Smalltalk MVC[1], Zeus[24] and Interviews[25] views, and ALV[6] link relationships, which use method arguments and constraints to maintain related object consistency. Change descriptions can document much more complex component updates than the “attribute updated” messages employed in these models, can be arbitrarily manipulated, and allow more flexible responses during propagation along relationships.

Our reusable object-oriented framework allows programmers to build new applications without having to implement large amounts of consistency management code. CPRGs are, however, less abstract than constraint-based languages and attribute grammars, such as the Cornel Program Synthesizer[10] and Garnet[5]. Programmers must specialise various framework classes to describe interobject dependencies, rather than just write simple constraint equations. CPRGs can, however, be used to implement applications with much more flexible consistency mangement systems than these other approaches.

As CPRG change descriptions generally hold before- and after-update values, the CPRG response mechanism is more powerful than most constraint and attribute grammar mechanisms. Component’s can determine their response using information about both their current and previous states, difficult to do in purely attribute constraint-based systems. This is, however, lower-level and more “operational” in nature than most constraint systems, where the consistency management algorithm is completely defined by the system, and not usually modifiable by programmers.

Components can generate, store and manipulate change descriptions in arbitrary ways. This contrasts to other systems which generate change description-like objects for use by undo/redo mechanisms, such as Unidraw[3], lazy consistency management, such as Object Dependency Graphs[16], and history items, such as ItemLists[4]. All of these systems store their equivalents to change descriptions internally and do not allow programmers to make use of them in any easy way. With CPRGs, programmers can also have components generate and store programmer-defined change descriptions, which are handled in the same way as those built into the CPRG architecture. This is useful representing arbitrary events not directly related to component state changes.

The inconsistency handling mechanism of[17] uses viewpoints and logic to record inconsistencies between different views of software development. CPRGs record inconsistencies in a more flexible way by storing change descriptions describing the inconsistency. At a later date these can be either actioned by the environment to try and achieve consistency, or presented to users to manually establish consistency, as done for some MViewsDP and SPE textual view updates.

10. CONCLUSIONS

Many software applications require diverse consistency management techniques to be employed. Change Propagation and Response Graphs (CPRGs) allow such systems to be more easily designed and implemented by providing a set of reusable abstractions for implementing a wide range of consistency

management systems. The kernel CPRG model represents software system data as attributed graph components and relationships, operations on which result in discrete change descriptions being generated. Propagation, response to, and storage of these change descriptions supports inter-object attribute constraint and recalculation, multiple textual and graphical views with flexible view consistency, an undo/redo facility, versioning and CSCW support, and change description composition. These diverse techniques are supported in a homogeneous way by CPRGs, and can be easily mixed, unlike most comparable software architectures for consistency management. Implementation and reuse of an object-oriented framework of classes for the CPRG model has demonstrated it has a wide range of uses and can be efficiently implemented.

The CPRG framework is rather low-level, requiring programmers to specialise framework classes to describe data dependencies. We are currently exploring ways of using the notion of discrete change propagation from with a programming language, in conjunction with higher-level constraint and attribute grammar specifications. This will allow CPRGs to form the implementation basis of systems, with flexible change description response still supported. Other, more abstract, consistency management techniques, such as attribute grammars and constraint equations, will be automatically translated into a CPRG implementation.

REFERENCES

1. G.E. Krasner and S.T. Pope A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (1988), 8-22.

2. S.P. Reiss GARDEN Tools: Support for Graphical Programming. In *Proceedings of Advanced Programming Environments,* Lecture Notes in Computer Science #244, Springer-Verlag, Trondheim, Norway, June 1986, pp. 59-72.

3. J.M. Vlissides and M. Linton Unidraw: A framework for building domain-specific graphical editors. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software and Technology,* ACM Press, 1989, pp. 158-167.

4. R.B. Dannenberg A Structure for Efficient Update, Incremental Redisplay and Undo in Graphical Editors. *Software-Practice and Experience 20*, 2 (February 1990), 109-132.

5. B.A. Myers Garnet: Comprehensive Support for Graphical, Highly Interactive User Interfaces. *COMPUTER 23*, 11 (1990), 71-85.

6. R.D. Hill The Abstraction-Link-View Paradigm: Using Constraints To Connect User Interfaces to Applications. In *Proceedings of CHI '92: Human Factors in Computing,* ACM Press, 1992, pp. 335-342.

7. J.C. Grundy *Multiple textual and graphical views for Interactive Software Development Environments*, Ph.D. dissertation, University of Auckland, Department of Computer Science, June 1993.

8. B. Backlund, O. Hagsand, and B. Pherson Generation of Visual Language-oriented Design Environments. *Journal of Visual Languages and Computing 1*, 4 (1990), 333-354.

9. S.E. Hudson Incremental Attribute Evaluation: A Flexible Algorithm for Lazy Update. *ACM TOPLAS 13*, 3 (July 1991), 315-341.

10. T. Reps and T. Teitelbaum Language Processing in Program Editors. *COMPUTER 20*, 11 (November 1987), 29-40.

11. J. Vlissides *Generalized Graphical Object Editing*, Ph.D. dissertation, Stanford University, 1990.

12. M. Ratcliffe, C. Wang, R.J. Gautier, and B.R. Whittle Dora - a structure oriented environment generator. *IEE Software Engineering Journal 7*, 3 (1992), 184-190.

13. S.P. Reiss Working in the GARDEN Environment for Conceptual Programming. *IEEE Software 4*, 11 (November 1987), 16-26.

14. T.R. Henry and S.E. Hudson Using Active Data in a UIMS. In *Proceedings of the ACM SIGGRAPH Symposium on User Interface Software,* 1988, pp. 167-178.

15. J.C. Grundy, W.B. Mugridge, J.G. Hosking, and R. Amor Support for Collaborative, Integrated Software Development. In *Proceedings of the 7th Conference on Software Engineering Environments,* April 5-7, Netherlands, IEEE CS Press, 1995.

16. M.R. Wilk Change Propagation in Object Dependency Graphs. In *Proceedings of TOOLS US '91,* Prentice-Hall, August 1991, pp. 233-247.

17. A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Ssoftware Engineering 2*, 8 (August 1994), 569-578.

18. G. Avrahami, K.P. Brooks, and M.H. Brown A Two-View Approach to Constructing User Interfaces. *ACM Computer Graphics 23*, 3 (1990), 137-146.

19. J.C. Grundy and J.G. Hosking A framework for building visusal programming environments. In *Proceedings of the1993 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1993, pp. 220-224.

20. R. Amor, G. Augenbroe, J.G. Hosking, W. Rombouts, and J.C. Grundy, "Directions in modelling environments," , Dept of Civil Engineering working paper, 1993.

21. P. Lyons, C. Simmons, and M. Apperley HyperPascal: Using visual programming to model the idea space. In *Proceedings of the13th New Zealand Computer Society Conference,* Auckland, August 1993, pp. 492-508.

22. J.C. Grundy, J.G. Hosking, S. Fenwick, and W.B. Mugridge *Visual Object-Oriented Programming,* M. Burnett, A. Goldberg, T. Lewis Eds, Manning/Prentice-Hall (1994), Chapter 11.

23. J.G. Hosking, S. Fenwick, W.B. Mugridge, and J.C. Grundy, "Cover yourself with Skin," Technical Report, Department of Computer Science, University of Auckland, 1994.

24. M.H. Brown Zeus: A System for Algorithm Animation and Multi-View Editing. In *Proceedings of the 1991 IEEE Symposium on Visual Languages,* IEEE Computer Society Press, 1991, pp. 4-9.

25. M. Linton, J.M. Vlissides, and P.R. Calder Composing User Interfaces with InterViews. *COMPUTER 22*, 2 (1989), 8-22.