Keeping things consistent

JOHN HOSKING, WARWICK MUGRIDGE and ROBERT AMOR

Department of Computer Science
University of Auckland
Private Bag 92019, Auckland
{john,rick,trebor}@cs.auckland.ac.nz

JOHN GRUNDY

Department of Computer Science
University of Waikato
Private Bag 3105, Hamilton
jgrundy@cs.waikato.ac.nz

Abstract

In many applications there is a need to keep shared or related information consistent. For example, in a CADrafting package it is necessary to keep views of an artefact from different perspectives consistent with one another. In software development environments there is a need to keep diagrammatic views, such as OO analysis diagrams, consistent with corresponding textual code. In this paper, we describe a number of techniques we have developed to solve consistency problems in a wide variety of application domains. These techniques include uni- and bi-directional constraints, propagation of discrete change descriptions, and transaction based version merging.

1. Introduction

Software designers are often faced with a need to keep different parts of an application consistent with one another as changes are made to the program or system state. These can be quite low-level, such as ensuring the rendered image of a figure moves by the appropriate amount in response to a drag event, through to very high level, such as ensuring a system implementation is kept consistent with changes to its requirements or design.

In this paper, we describe a number of approaches to maintaining consistency across applications that we have developed over the years, together with some of the motivation for their development. We commence with a description of Kea, which provides one way dependencies as a way of keeping the results of function applications consistent with changes to input data without programmer overhead. This is followed by a description of multi-directional constraints as implemented in the Snart object-oriented constraint language. Such constraints permit multiple representations of information to be kept mutually consistent with one another in a declarative manner. We then introduce the CPRG (Change Propagation and Response Graph) model. This elevates the description of changes to first class status, allowing complex reasoning about changes, and permitting implementation of partial consistency and change histories. This is followed by a description of VML (View Mapping Language), which provides a somewhat more declarative approach than the CPRG model to handling complex mappings between large, related models. We conclude with a brief description of related work and our future directions.

2. Kea and functional consistency

Kea is a functional object-oriented language (Hamer, 1991). Its motivation was to provide a means of expressing code of practice provisions in a manner allowing their interpretation for conformance checking. Several such applications have been developed for interpreting building codes, in collaboration with the Building Research Association of New Zealand. These include FireCode (Hosking et al, 1987), Seismic (Hosking et al, 1989), WallBrace (Mugridge and Hosking, 1988), and ThermalDesigner (Amor et al, 1992). Fig. 1 shows a typical provision, taken from the thermal insulation code of practice on which ThermalDesigner is based.

```
The seasonal heat loss of a building
is the air heat loss plus the sum of
the floor, wall, window, and roof
heat losses for each space in the
building.
The heat loss of a floor is the floor
area times the annual loss factor of
the floor divided by the thermal
resistance of the floor.
```

Figure 1: Example provision used in the development of ThermalDesigner

In typical usage, such provisions are interpreted in a functional manner (Fenves et al, 1987) as opposed to a multi-way constraint. For this reason, Kea uses a functional representation of provisions. In addition, a code conformance system requires a representation of the building being checked. Kea uses an object-oriented approach, which is the norm in code interpretation systems (de Waard, 1992). The code provision representation is then incorporated with the representation by integrating the code provision functions within the object-oriented building model.

As an example, Fig. 2 is a Kea representation of the provision of Fig. 1 and its implicit building model. Classes represent buildings, spaces, floors, etc. Buildings contain a collection of Spaces (indicated by the spaces attribute of Building), while Spaces contain collections of Floors, Walls, Windows, and Roofs. The provision is represented by a function seasonal_heat_loss in class Building, which makes use of functions in the other classes (eg heat_loss in Space). Liberal use is made of Kea's list manipulation functions which allow projection of values from lists (eg collect), accumulation of values from a list (eg sum), and mapping of functions over lists.

I/O facilities are also needed for entering building information such as plan and materials data. Kea handles this via PlanEntry, a specialised plan drawing system (written in C), plus a forms interface that allows objects to be associated with forms and object attributes with menus, edit fields etc. Thus information for checking conformance of the seasonal_heat_loss provision is obtained from PlanEntry data supplemented by form data (see Fig. 3). Provision calculation is initiated by ThermalDesigner constructing a form requiring (directly or indirectly) the provision's results. This in turn triggers access to the PlanEntry data, and construction of forms for supplementary data.

In design-assistance type applications, such as ThermalDesigner, it is important to allow users to experiment with modifications to their designs. In a conventional application, this requires large amounts of code to handle modification of input data and appropriate recalculation of results. Kea takes a very different approach to handling such modifications. Each Kea function application also implicitly defines a one way constraint between its result and the values used in deriving that result. Any change to one of those latter values causes recalculation of the function result (with caching of intermediate results for efficiency). Thus, changing an input field on a form, or an element of the plan in PlanEntry, automatically causes recalculation of any values dependent on that data item.

Object creation and destruction can be conditionally dependent on the values of input data items. Thus, changing an input data value may cause new objects to be created and old objects destroyed (together with their associated forms) to maintain consistency with the new value. The dependency mechanism also works over list operations, such as the list accumulations used in the seasonal_heat_loss calculation of Fig. 2. Thus, the addition of a new Space using PlanEntry will trigger recalculation of seasonal_heat_loss due to the change in the Building's spaces attribute.

```
class Building
  spaces: list Space.
  public seasonal_heat_loss: float
    := air_heat_loss + sum(collect(s
       in spaces, s^heat_loss)).
  air_heat_loss: float := ...
end Building.

class Space
  floors:list Floor.  walls: list
Wall.
  windows:list Window. roofs: list
Roof.
    public heat_loss: float
       := floor_heat_loss +
         wall_heat_loss +
         window_heat_loss +
      roof_heat_loss.
    floor_heat_loss: float :=
       sum(collect(f in floors,
          f^heat_loss)).
       :  % detailed code elided
end Space.

class Floor
 public heat_loss: float := area *
          annual_loss_factor /
          thermal_resistance.
  annual_loss_factor: float := ...
       :  % etc
end Floor
```

Figure 2: Kea representation of the provision of Fig. 1.

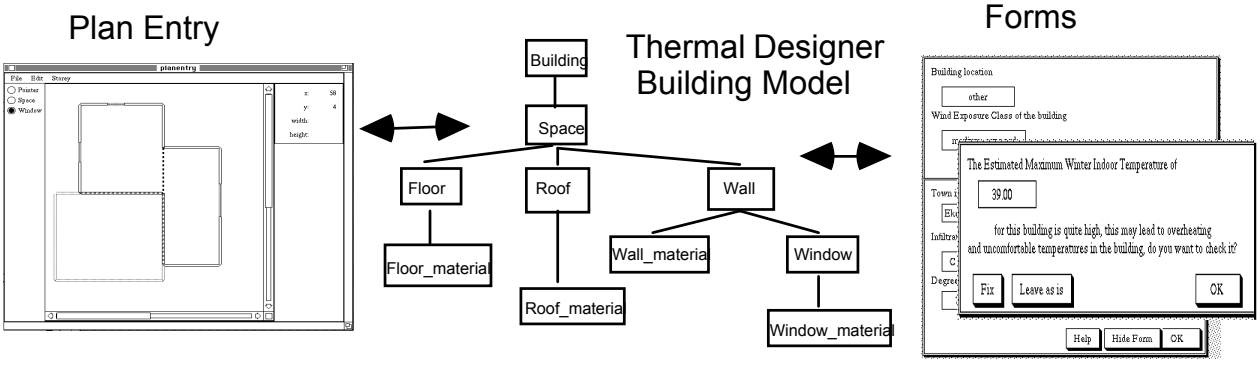


Figure 3: ThermalDesigner framework

This approach to consistency is in marked contrast to the typical procedural approach. In writing a Kea program, the programmer constructs programs as if input data, once supplied, remains unchanged. Thus *there is no explicit code required to maintain consistency* when input values do change; the effects are instead calculated automatically. Thus the programmer can concentrate on writing a "clean" declarative representation of the code of practice provisions, without worrying about the effects of modifications to the program state.

Kea's one way dependency mechanism is similar to the automatic recalculation facilities of spreadsheets. Kea differs in that the dependency is set up from the tree of function applications, rather than through statically specified cell equations, and the dependencies are distributed over all of the objects, forms, and PlanEntry windows involved.

Kea provides a relatively clean approach to consistency maintenance. However, there is a price to pay in terms of the limited expressiveness of the language. The functional nature of Kea, while being exceptionally good for expressing code of practice provisions, makes it difficult to direct the flow of control. This makes issues such as directing a dialogue, or managing inter-form navigation difficult to do within the language. As a result Kea has been extended with procedures and conditional procedures (triggered by changes to data values). This allows sequences of actions, in particular the hiding and displaying of forms, to be executed. The addition of these language features, however, considerably complicates an otherwise relatively clean and elegant architecture for managing the one way dependencies.

A more serious problem, though, is the inability of the Kea one way dependency approach to handle bi-directional consistency. For example, in the PlanEntry component it is desirable to have multiple editable views of a building (eg plan and elevation) with editing changes in either view reflected in the other (and of course in the underlying building model). Keeping such views mutually consistent is beyond the ability of Kea's one way dependencies. A similar problem arises with multiple data entry points when integrating tools for code conformance checking (Mugridge and Hosking, 1995). To provide a solution to this problem alternative techniques, such as those described in the next three sections, are needed

3. Snart and multi-directional constraints

Snart (Hosking et al, 1994) is an object-oriented extension of Prolog with multi-directional constraint propagation (Hill, 1993; Freeman-Benson, 1991). The imperative features of Snart and Prolog provide general-purpose facilities for procedural programming within an object-oriented setting. This, for example, permits all user-interface processing to be developed within the language, rather than having to use a separate language (as with Kea).

The addition of multi-directional constraints to Snart was motivated by the multi-view plan and data entry problems described above. We felt that a multi-directional constraint approach would allow changes to be made at any of several inter-dependent points and propagated to the others.

Fig. 4 shows the use of multiple views in a version of PlanEntry implemented in Snart (Hosking et al, 1994). Tools for creating spaces, roofs, walls and windows are provided. Handles are used to manipulate plan elements. Equality constraints may be placed between the handles of spaces. For example, two spaces may be constrained to remain abutted, so that moving (or resizing) either space will move the other. A labelled *view line* in a view indicates the plane of another orthogonal view; a view line may be moved to provide a different view or may be constrained to be aligned to a surface of one of the spaces making up the building. Additional views may be added as desired.

In PlanEntry, multi-directional constraints are used for defining inter-dependencies between building elements (such as abutment) and also to maintain consistency between multiple views of the building. This is illustrated at the bottom of Fig. 4, where the objects representing a plan view and a view from the left of a building space are shown. A change to the 2-D representation in one view is propagated by multi-directional constraints to an underlying 3-D model of the building; this change is then propagated to any other views affected by the change.

As a more specific example, consider roofs. The *z* dimension of a roof is represented by three attributes (Fig. 5(a)). A change to any one of these attributes leads to a change to one of the others. For example, if the top of the roof (*z1*) is dragged upwards, the *height* is changed. The relationship between these attributes is handled by the constraint *height = z1 - z* in class *roof* (Fig. 5(b)). When the value of any one of *height, z,* or *z1* changes, one of the other attributes is changed in order to resatisfy the equality specified by the constraint.

In general, when there are more than two attributes in a constraint it is unclear how the change is to be handled. This problem is solved in Snart with *directions*. A constraint may have explicit directions that specify how the constraint is to be interpreted when an attribute value is changed. Default directions are applied if none are specified. The defaults are as follows: a change to the first attribute leads to constraint propagation to the second; any other changes lead to the first attribute being recalculated. Thus, the default directions for the constraint in Fig. 5 are: (height=>z1, self=>height, z=>height, z1=>height) where (height=>z1) means that on a change to height the attribute z1 is recalculated.

The direction (self=>height) means that the height will be calculated (if possible) when the constraint is first imposed.

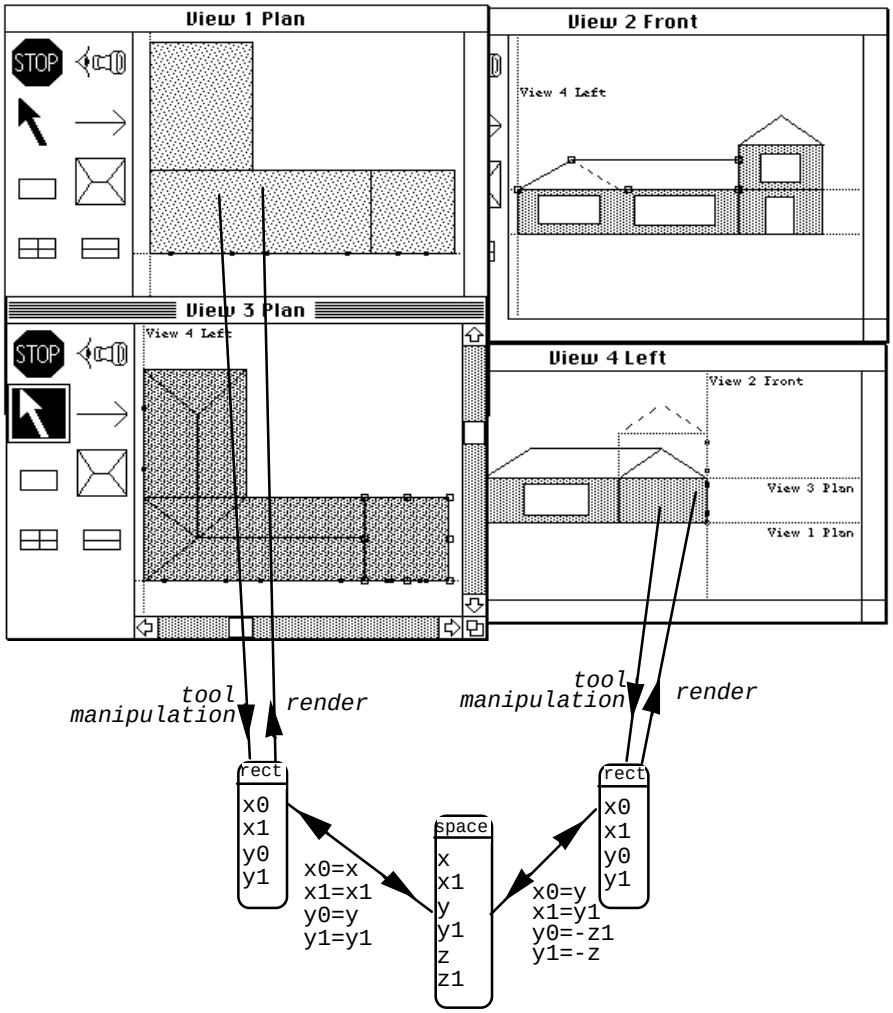
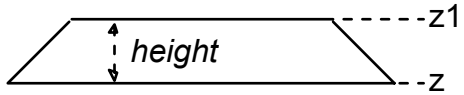


Figure 4: PlanEntry system in use




```
class(roof,
    features(height: int, z: int,
             z1: int, ...),
    ...
    constraints(height = z1 - z
           and_with move(z =>z1))
).
```

Figure 5: (a) Roof *z* dimension attributes (b) Snart constraint on roof attributes

The interpretation of a constraint can also depend on the *reason* for a change (as specified at the time an assignment is made). The constraint in Fig. 5 has an additional direction (following *and_with*), move(z=>z1), which means that if the current *reason* is *move*, a change to z will lead to z1 being recalculated (z1 := height + z). The overall effect of the constraint is that the top and bottom of the roof can be changed independently, with *height* automatically recalculated each time, but if the bottom of the roof is *moved* the top of the roof is also moved, keeping *height* constant.

Snart *demons* are used in PlanEntry to ensure that a changed building component is rendered afresh. A demon is a piece of Prolog code that is executed whenever the value of any of the Snart class attributes used in the demon is changed. In that respect they are very similar to the conditional procedures introduced into Kea. In a conventional language, procedural code would be needed to handle this dependency explicitly.

Snart has other features for constraint processing, including path handling, constraints over classifiers and objects, and relational constraints (Mugridge et al, 1995). However, it lacks one feature available in Kea which is particularly valuable for our work on code conformance systems: constraints over collections, such as the implicit 1-way dependencies over the accumulations in Fig. 2. Current work aims to add uni-directional collection constraints to Snart to rectify this.

4. Change Propagation and Response Graphs

CPRGs (Grundy et al, 1995b) provide an alternative consistency management approach to Kea's 1-way dependencies and Snart's n-way constraints. CPRGs represent information as attributed components linked by relationships (relationships are themselves components, ie. they are able to be linked by other relationships. Whenever a component is modified, a change description is generated documenting the exact change in the component's state. Thus, if a component Comp has its attribute *Name* changed from *John* to *Rick*, a change description of the form *update(Comp,Name,John,Rick)* is generated. Change descriptions may be low-level, documenting simple attribute changes, or may describe a more abstract change in the component's state.

Change descriptions are propagated to all relationships that the updated component participates in. These relationships then interpret the change description and either (i) update their own or related component states, (ii) propagate the change description to related components, or (iii) ignore the change in state of the updated component.

A simple example is shown in Fig. 6. This shows a dialogue editor and its corresponding CPRG components, representing the state and structure of the dialog and its controls. The dialog box border has been interactively dragged to a new location and the dialog's associated components must then be updated to move with the dialog box they belong to: 1) The drag causes the *dialog* CPRG component's X and Y co-ordinates to be changed; 2) *dialog* propagates change descriptions to its `parts` relationship, which forwards them to the *name* and *age* edit field controls; 3) *name* and *age* respond to the change descriptions by updating their own co-ordinates, to be consistent with that of *dialog*'s (ie. *name* and *age* are constrained to follow *dialog*'s positional changes); 4) *name* propagates the change descriptions its operations produced to *parts*, which propagates them to *dialog*. *dialog* ignores the changes in *name*'s state, as it is not constrained to follow its fields' positional changes. The graphical or textual renderings of the changed CPRG components are redrawn, to reflect their changed states (using an approach similar to that of Snart demons).

Programmers using CPRGs must define appropriate responses to change descriptions using operational-style code, in contrast to Kea and Snart constraints, which use functional and dependency-driven recalculation. CPRGs are thus less declarative than the Kea and Snart approaches, but more flexible, as programmers can define complex structures and responses to changes based on before/after update information.

CPRGs support several other consistency management techniques in addition to the simple attribute dependency described above. One powerful abstraction is generic relationships, which encapsulate a predefined set of change description responses. Examples include relationships implementing general graphical figure constraints (as above), aggregation structures, multiple views of information, and flexible attribute recalculation.

MViews, a framework for building multi-view editing environments, is based on the CPRG architecture (Grundy and Hosking, 1993). MViews provides extra abstractions for specifying base information repositories, multiple views of these repositories, and graphical and textual renderers and editors for manipulating view information. MViews has been used to implement several multi-view editing environments, including: MViewsDP, a dialogue box editor; MViewsER, an ER diagram/relation schema editor; EPE (Amor et al, 1995), an environment for EXPRESS programming; Cerno (Fenwick et al, 1994), a visualisation tool; and SPE (Grundy et al, 1995a). A screen dump from SPE (Snart Programming Environment) is shown in Fig. 7. SPE supports multiple textual and graphical views of a Snart program. These include graphical analysis and design, and textual implementation and documentation views. All of these views are kept consistent with one another via the CPRG consistency management mechanism.

CPRGs not only support consistency management, but also partial consistency and inconsistency management. For example, some SPE view updates can not be directly translated by SPE into changes to other affected views. A good example is when a client-supplier (method-calling) relationship is added to an SPE design view. This cannot be automatically added to textual implementation views, as SPE doesn't know the correct arguments and position for the method call in the textual code. CPRGs allow such partial inconsistencies to be documented and presented to users. In SPE this is done by inserting readable change descriptions into the textual code views to inform programmers of the graphical view updates.

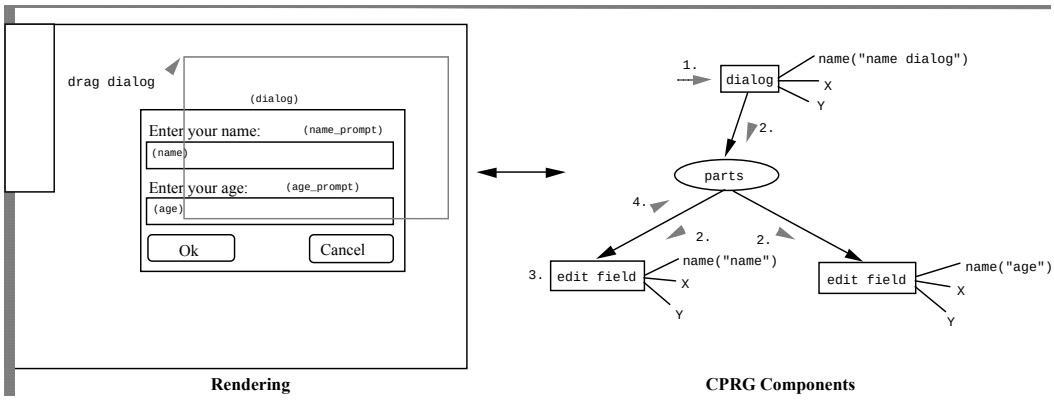


Figure 6: An example of the CPRG approach to consistency management.

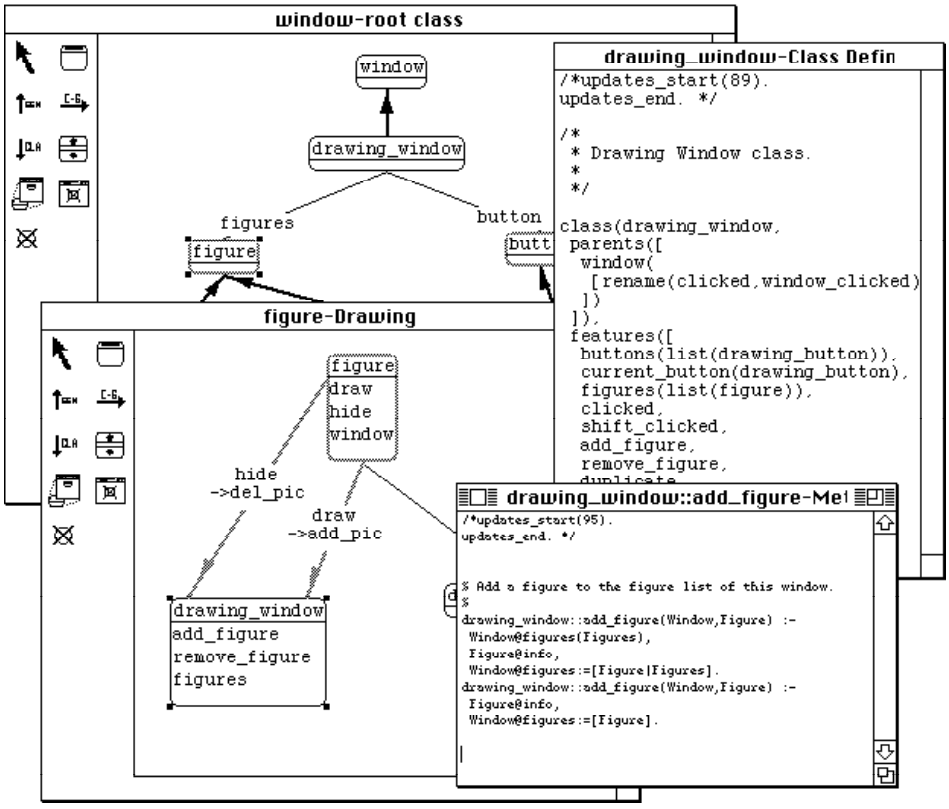


Figure 7: A screen dump from SPE (Snart Programming Environment).

Change descriptions can be stored by CPRG components, as they are first-class objects. These stored change descriptions can then be presented to users at a later date (to inform them of possible inconsistencies), can be stored to document the modification history of components, or can be used to support version control and Computer-Supported Collaborative Work systems (Grundy et al, 1995c). C-MViews extends MViews to support version control for views and components and to support both semi-synchronous and synchronous forms of collaborative work. C-SPE supports collaborative software development in Snart by reusing these extended CPRG facilities.

The CPRG architecture is currently implemented as a framework of Snart classes (not using the constraint extensions described above). Users of CPRGs specialise these classes to built new applications incorporating CPRG consistency management techniques. Several kinds of generic relationships are provided to assist programmers in building new applications. So far, CPRGs have mainly been used in the construction of MViews. We are currently investigating incorporating CPRG-style event constraints into Snart, to be used in conjunction with Snart's constraint system.

CPRGs are, in many ways, a lower-level mechanism than the Snart or Kea constraint approaches, requiring more operational code for their implementation. However, they are considerably more flexible, permitting partial consistency and inconsistency to be represented sensibly as well as having the ability to store change descriptions to permit reasoning about changes over time. All of these are difficult, if not impossible, with the Kea and Snart approaches. Despite being low-level, the ability to package change responses into reusable relationship classes permits quite high-level consistency tools to be developed and reused.

5. VML

In contrast to the fairly low-level, operational nature of CPRGs, VML (View Mapping Language) is a high level, declarative language suitable for describing correspondences between two complex models (Amor, 1994; Amor and Hosking, 1995). VML was developed for use in the domain of architecture and engineering design support. Models in this domain are characterised by their large size, complexity of data structures, and their individuality. The individuality results from each design tool (and user) focusing on one small portion of the domain and attempting to minimise the amount of data entry to its model. As a result, structures are developed which are unique to the tool or subdomain. The architecture and engineering domain is also characterised by having many designers working concurrently on portions of the design. When a designer completes a design task it must be passed to all other designers whose work is affected by the task so the modifications can be taken into account in the design tasks they are performing.

In a VML environment it is assumed that all mappings will be between two models. As far as practicable a VML definition treats both models as equal partners in a mapping. When it is necessary to map information between several models and a single central model each mapping is specified independently so that the mapping implementation can manage updates to and from each model individually. A mapping is described through a set of *inter_class* definitions which specify: the classes involved in the mapping; the conditions (invariants) under which the mapping may take place; the attribute mappings that can be applied; and initial values to be set on object creation. Fig. 8 shows a single *inter_class* definition. This specifies the relationship between *trombe_wall* entities in one model and *trombe_wall* and *trombe_type* entities in another. The invariant specifies that the mapping relates *trombe_wall* objects with a particular *trombe_type* value X with *trombe_type* objects with *name* equal to X. Equivalence mappings are specified via constraint equations (eg *area = height*width*), functions, or procedural code. Where procedural code is used, two mappings must be specified to be able to move data in each direction between the views.

VML also has a graphical notation for specifying a mapping. Figure 9 show a graphical view of the trombe_wall *inter_class* definition. The graphical notation provides for high level views of the connections between objects while the textual notation details the exact connection between classes and their attributes. Thus, in Fig. 9, the equivalences are indicated iconically by type (direct equality, or constraint equation in this case), with links to the attributes involved in each equivalence or invariant.

A mapping system manages interaction between object stores representing instances of the models specified in a VML mapping. The mapping system uses a sophisticated transaction-based approach to maintain consistency between the object stores. A transaction denotes a portion of work completed in one view (where a view is one user's model of a building defined using the structures specified in a schema). For example, a transaction could be the initial layout of a building, the results of a simulation, or the results of actioning a change request. The granularity of transaction size for a view is under user control, and can vary from coarse, such as the complete specification of a building's layout, to fine, such as a change to an individual attribute.

Whenever a user has finished a particular task, or a design tool has finished its calculations, a *transaction completion record* is created. This is passed to all mapping handlers managing mappings between the modified object store and other object stores. The mapping handler action is dependent on the type of mapping specified. A read-only mapping handler discards the transaction as there is no authority to update the other object store. A writable mapping handler applies the modifications described in the transaction via the mapping equations to the other view. However, if the other data store is an integrated view (ie a central view that coordinates data for several users and applications), a check is made that the sending store is up to date with all other transactions applied to the integrated view. If this is not the case, the model that the transaction was created in is inconsistent with the integrated model. All outstanding transactions from the integrated view must first be applied to the data store before it may pass its transaction back to the integrated store.

```
inter_class([trombe_wall],[trombe_wall, trombe_type],
    invariants(trombe_wall.trombe_type = name),
    equivalences(area = height * width,
           glazing = glazing,
           vent_area = sum(vents=>(height * width)),
           trombe_type = trombe_type,
           perf_ratio = perf_ratio)
    ).
```

Figure 8: A lexical VML specification

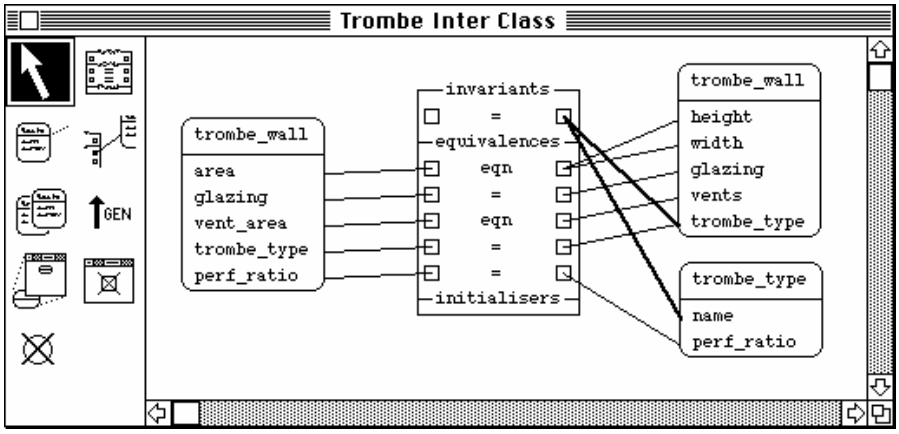


Figure 9: A graphical VML specification

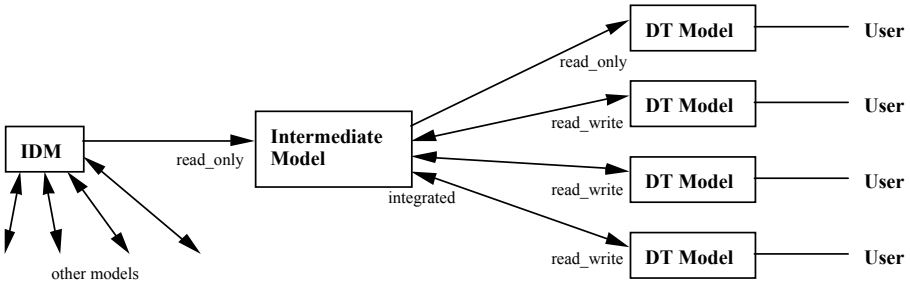


Figure 10: Mixing mapping types to isolate modifications

Being able to define the mapping type in the VML specification allows a network of connections between object stores with object stores playing different roles based on the mapping type. For example Fig. 10 shows the use of an Intermediate Model (IM) which has a read-only mapping to a shared Integrated Data Model (IDM). The IM is then attached to a number of tool models, acting as an integrated model for these tools. Thus these tools can interact with the IM, and hence, effectively, with one another without affecting the IDM. This permits experimentation on the IDM to occur, without affecting the IDM itself as changes to the IDM are reflected in the IM but not vice-versa.

6. Discussion

We have described a number of approaches to maintaining consistency in software systems. Each approach offers a more declarative and structured approach to the problem than do more conventional solutions. In addition, each of the approaches can be seen as a means of implementing systems designed using the *tool-abstraction* paradigm (Garlan et al, 1992).

The systems we have described are, by no means, the only declarative or structured approaches to consistency, but are, rather, representatives of the different styles of consistency management. For example, the 1-way dependency approach of Kea has already been compared to the automatic recalculation facilities of spreadsheets. The Garnet system (Myers, 1990) also implements 1-way constraints and has a more comprehensive approach to user interface development than does Kea.

ThingLab (Borning, 1981), Rendezvous (Hill, 1993) and Kaleidoscope (Freeman-Benson, 1991) are all Object-Oriented languages incorporating perturbation-based multi-directional constraints like Snart. However, none provide the degree of flexibility and control over constraint processing as does Snart. Much work has focussed on the multiple view consistency problem, with the MVC approach developed for Smalltalk being the most widely adopted (Krasner and Pope, 1988).

Variations on this approach which have similarities to CPRGs include ALV's bidirectional constraints (Hill, 1992), the ItemList structure (Dannenburg, 1990), Object Dependency Graphs (Wilk, 1991) together with multi-view editing systems such as LOGGIE (Backlund et al, 1990) and Unidraw (Vlissides and Linton, 1989).

Recently there have been many projects which have looked at formalising the specification of complex mappings through the use of specialised mapping languages (eg. Bailey 1994; Hardwick 1994; Clark 1992). Unlike VML, languages developed in these projects are basically procedural in nature and provide a very low level view of the correspondence between two models. These new languages also follow the form of relational database views, so that where the mappings need to be bi-directional, it is necessary to specify a separate mapping definition for each direction.

Current work aims to unify the approaches we have developed, to provide systems supporting several consistency paradigms within the one language/framework. For example, Snart is being extended to incorporate 1-way dependencies over list operations, similar to those of Kea. CPRG-like extensions to Snart have been proposed which incorporate propagation of events as well as data. These extensions would considerably simplify CPRG frameworks, such as MViews. The C-MViews extensions of the CPRG model have much in common with the VML transaction management system and unification of these approaches should also be possible.

Acknowledgments

The authors acknowledge the support of the Auckland University Research Committee, The Building Research Association of New Zealand, and the Foundation for Research Science and Technology in pursuing this research.

References

Amor, R.W., Hosking, J.G., Mugridge, W.B., Hamer, J., Williams, M., 1992: ThermalDesigner: an application of an object-oriented code conformance architecture. *Joint CIB Workshops on Computers and Information in Construction,* CIB Publication 165, Montreal, Canada, pp 1-11.

Amor, R., Augenbroe, G., Hosking, J., Rombouts, W. and Grundy, J., 1995: Directions in Modelling Environments, accepted for publication in *Automation in Construction*.

Amor, R., 1994: A Mapping Language for Views, Department of Computer Science, University of Auckland, Internal Report, 30pp

Amor, R.W., and Hosking, J.G., 1995: Mappings: the glue in an integrated system, accepted for publication in *Procs 1st European Conference on product and process modelling in the building industry*.

Bailey, I., 1994: *EXPRESS-M Reference Manual, Product Data Representation and Exchange*, ISO TC184/SC4/WG5 N51, 66p.

Backlund, B., Hagsand, O., and Pherson, B., 1990: Generation of visual language-oriented design environments, *Journal of Visual Languages and Computing* , 1(4), pp333-354.

Borning A, 1981. The programming language aspects of ThingLab, a constraint-oriented simulation laboratory, *ACM Trans. Programming Languages and Systems*, 3(4), pp353-387.

Clark, S.N., 1992: Transformr: *A Prototype STEP Exchange File Migration Tool*, National PDES Testbed Report Series, NISTIR 4944, US Department of Commerce, National Institute of Standards and Technology, 13pp

Dannenberg, R.B., 1990: A structure for efficient update, incremental redisplay and undo in graphical editors, *Software-Practice and Experience*, 20(2), pp109-132.

de Waard, M, 1992: *Computer Aided Conformance Checking*, Thesis Technische Universiteit Delft

Fenves, S J, Wright, R N, Stahl, F I and Reed, K A, 1987: *Introduction to SASE:Standards Analysis, Synthesis, and Expression*, NBSIR 87-3513, National Bureau of Standards.

Fenwick, S., Hosking, J., and Mugridge, W., 1994: A visualization system for object-oriented programs, in Mingins, C. and Meyer, B. *Proc TOOLS 15*, Prentice Hall, Sydney, pp93-103.

Freeman-Benson, B, 1991. *Constraint imperative programming*, Tech. report 91-07-02, University of Washington.

Garlan, D., Kaiser, G.E., and Notkin, D., 1992: Using tool abstraction to compose systems, *COMPUTER*, 25(6), pp30-38

Grundy, J.C. and Hosking, J.G., 1993: Constructing multi-view editing environments using MViews, *Proc. 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, Los Alamitos, CA, pp220-224.

Grundy, J., Hosking, J., Fenwick, S., and Mugridge, W., 1995a: Connecting the pieces, Chapter 11 of *Visual Object-Oriented Programming*, Margaret Burnett, Adele Golberg, and Ted Lewis (eds), Manning Publications, Greenwich, CT, USA, pp229-254.

Grundy, J., Hosking, J., and Mugridge, W., 1995b: Supporting flexible consistency management via discrete change description propagation, submitted to *Software Practice and Experience*.

Grundy, J., Mugridge, W.B. Hosking, J.G. and Amor, R.W., 1995c: Support for collaborative, integrated software development, *Proc 7th Conference on Software Engineering Environments*, The Netherlands, IEEE CS Press, pp84-94.

Hamer, J., 1991: *Kea 1.0 Reference Manual*, BRANZ Contract 85-024 Technical Report No. 20, Department of Computer Science, University of Auckland.

Hardwick, M., 1994: *Towards Integrated Product Databases Using Views*, Technical Report 94003, Rensselaer Polytechnic Institute, Troy, New York, USA, 18pp

Hill, R.D., 1992: The Abstraction-Link-View paradigm: using constraints to connect user interfaces to

applications, in *Proceedings of CHI '92: Human Factors in Computing,* ACM Press, pp335-342

Hill, RD, 1993. The Rendezvous constraint maintenance system, *UIST'93*.

Hosking, J.G., Mugridge, W.B. and M. Buis, 1987: FireCode: a case study in the application of expert systems techniques to a design code. *Environment Planning and Design B*, 14, pp267-280, 1987.

Hosking, J.G., Lomas, S., Mugridge, W.B., and A.J. Cranston, 1989: The development of an expert system for seismic loading. *Civil Engineering Systems*, 6 (1-2), pp 27-35.

Hosking, J.G., Mugridge, W.B., and Blackmore, S., Objects and constraints: a constraint based approach to plan drawing, in Mingins, C. and Meyer, B. *Proc TOOLS 15*, Prentice Hall, Sydney, pp9-19, 1994.

Krasner, G.E. and Pope, S.T., 1988: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk-80, *Journal of Object-Oriented Programming*, 1(3), pp8-22.

Mugridge, W.B. and J.G. Hosking, 1988: The development of an expert system for wall bracing design. *Proc. NZES'88 The Third New Zealand Expert Systems Conference*, Wellington, pp10-27.

Mugridge W.B. and Hosking J.G.: Towards a lazy evolutionary common building model, *Building and Environment*, 30, (1), pp99-114, 1995.

Mugridge, W.B. , Blackmore, S., Hosking, J.G., and Grundy, J.C., 1995: Dual-propagation and higher-level constraints in Snart, submitted to International Conference on Principles and Practice of Constraint Programming to be held in Marseille, September 1995.

Myers, B.A., 1990: Garnet: comprehensive support for graphical, highly interactive user interfaces, *COMPUTER*, vol. 23, no. 11, pp71-85.

Vlissides, J.M. and Linton, M., 1989: Unidraw: A framework for building domain-specific graphical editors, in *UIST,* ACM Press, pp158-167.

Wilk, M.R., 1991: Change propagation in object dependency graphs, *Proc TOOLS US '91,* pp233-247.