SUPPORT FOR CONSTRUCTING ENVIRONMENTS WITH MULTIPLE VIEWS

John C. Grundy†, John G. Hosking††, Warwick B. Mugridge††, and Robert W. Amor†††

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

Department of Computer Science††
University of Auckland
Private Bag, Auckland, New Zealand
{john,rick}@cs.auckland.ac.nz

Building Research Establishment †††
Bucknalls Lane, Garston,
Watford WD2 7JR, UK
trebor@bre.co.uk

ABSTRACT

We describe several approaches to supporting the construction of design environments with multiple views of common information. We then outline a new approach that draws on the best of each of them.

KEYWORDS

multiple views, view consistency, change descriptions, constraints, view mapping languages, process modelling

1. INTRODUCTION

The design of complex artefacts typically involves multiple designers sharing parts of an evolving design, using multiple design notations and tools. There can be much overhead in inter-notation translation and co-ordination of both design parts and designers to maintain overall design consistency. Automating the simpler aspects of these processes allows designers more time to resolve complex inconsistencies and to do creative design. We present several of our approaches to solving this problem in two application areas: the design of buildings and large software systems.

2. MVIEWS

MViews is a framework for constructing Integrated Software Development Environments (ISDEs) which support multiple textual and graphical views [8, 9]. MViews provides a general model for defining software system data structures and tool views, and a flexible mechanism for propagating changes between software components, views and tools. ISDE data is described by *components* with *attributes*, linked by a variety of *relationships*. Multiple views are supported by representing each view as a graph linked to the base software system graph. Each view is rendered and edited in either a graphical or textual form. Tools can be interfaced at the view level (as editors), via external view translators, or multiple base layers may be connected via inter-repository relationships [10]. Figure 1 shows the architecture of SPE (Snart Programming Environment), an MViews ISDE for object-oriented software development [11]. All basic graph editing operations generate *change descriptions* of the form: UpdateKind(UpdatedComponent, ...UpdateKind-specific Values...). For example, an attribute update is represented as: update(`Component,AttrName, OldValue,NewValue`).

Change descriptions are propagated to all components related to the updated one. Receiving components interpret descriptions and may modify themselves, producing further change descriptions. A wide range of facilities are supported, including attribute recalculation, bi-directionally consistent textual and graphical views, inconsistency recording, undo/redo, versioning, tool integration, and collaborative editing [12]. Developers specialise MViews framework classes to define software components, views and editing tools to produce a new ISDE. A persistent object store saves ISDE data.

We have built many environments with MViews [2, 11, 12, 13]. Several have been integrated together to produce multi-notational ISDEs such as OOEER [10], which keeps OOA/D and EER views fully consistent, and NIAMER [22], which keeps ER and NIAM views fully consistent. MViews ISDEs are more flexible than those of generator approaches [18, 20], and MViews has better abstractions for building multi-view editing systems than other framework approaches [5, 16, 19].

3. SNART CONSTRAINTS

Snart [15] is an object-oriented extension of Prolog with multi-directional constraint propagation. It provides imperative features, together with constraints to maintain consistency when data values change. Unlike MViews, constraints are specified in a declarative fashion, with change propagation and response handled automatically. PlanEntry, a plan drawing system [15], illustrates use of Snart constraints (Figure 2). PlanEntry provides multiple 2D views of a building. Constraints maintain consistency between these views and an underlying 3D representation of the building. Changes to views propagate through the underlying model and on to other views. Handles are used to manipulate plan elements and also to impose constraints between them; for example, two spaces may be constrained to remain abutted.

As a more specific example, suppose the *z* dimension of a roof is represented by three attributes, as in Figure 3(a), related by the constraint *height = z1 - z* in class *roof* (Figure 3(b)). If the value of any one of *height, z,* or *z1* changes, one of the other attributes is changed to resatisfy the equality. Thus, if the roof top (*z1*) is dragged upwards, the *height* is changed.

If there are more than two attributes in a constraint it is unclear which to change. This ambiguity is solved in Snart with *directions*. A constraint may have explicit directions specifying how it is to be interpreted on an attribute value change. Default directions apply if none are specified: a change to the first attribute leads to constraint propagation to the second; any other changes lead to the first attribute being recalculated. The roof constraint defaults are thus: (height=>z1, z=>height, z1=>height) where (height=>z1) means that on a change to height the attribute z1 is recalculated.

Figure 1. The SPE architecture as represented in MViews.

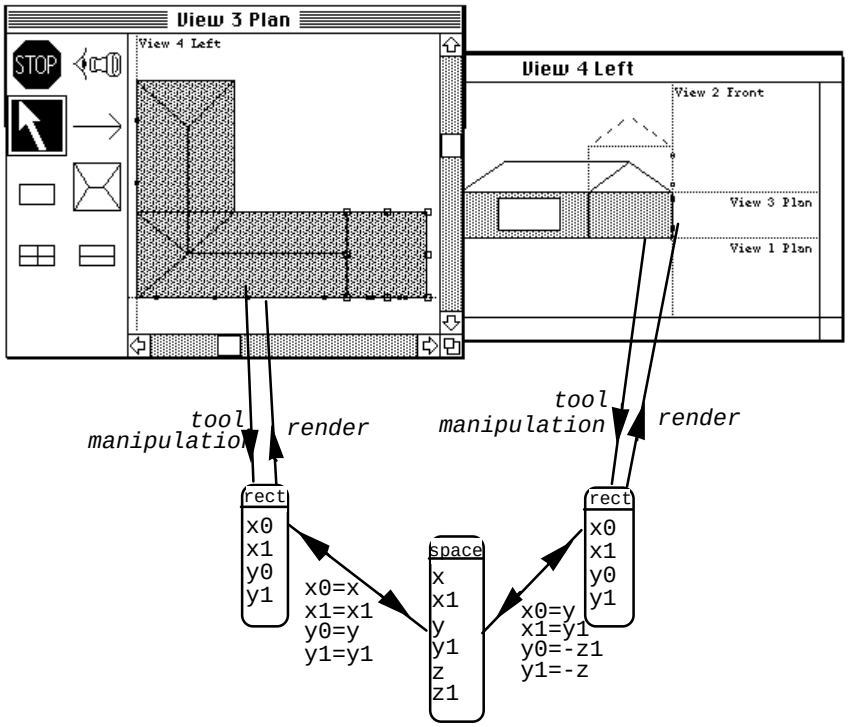
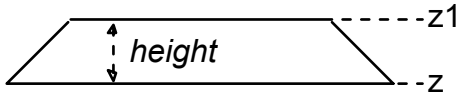


Figure 2: Multiple Views in PlanEntry




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

).
```

Figure 3: (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 (specified when an assignment is made). The constraint in Figure 3 has an extra direction *and_with move(z=>z1)*, meaning that if the current *reason* is *move*, changing z will lead to z1 being recalculated (z1 := height + z). The overall effect is that the roof top and base can change independently, with *height* automatically recalculated, but if the base is *moved* the top is also moved, keeping *height* constant. Directions and reasons provide a simpler disambiguation mechanism than hierarchical constraints as in Kaleidoscope [7].

Snart *demons* are used in PlanEntry to ensure a changed building component is re-rendered. A demon is a procedure executed whenever the value of any of the Snart object attributes used in the demon is changed. 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, constraints over collections of objects, and relational constraints [17].

4. VML

VML (view mapping language) [3] takes a more abstract approach than MViews or Snart. VML specifies bidirectional mappings between two schemas. The schemas can describe information models of two design tools, a design tool and a shared common data repository, or even two versions of the same model. The motivation is to support integration of building design tools via a common repository [1], but with explicit mapping specifications rather than ad-hoc techniques as used in other integration projects [4, 6]. VML thus differs from the previous two approaches in its primary aim of tool integration rather than tool construction.

A VML specification consists of several *inter_class* specifications, such as that in Figure 4. Each describes a mapping between objects of specific classes in one schema (*space_face*, *material_face*, *building* in Figure 4) with matching ones (*wall*) in another. *Invariants* specify applicability of an *inter_class* for a particular set of objects. In Figure 4, the *space_face location* must be *'ext'* and the *material_face type_of_face* must be *'wall'* and the *space_face* and *material_face* planes coincident for *inter_class* to be valid.

Equivalences specify mappings between attributes of the objects involved. These can be constraint-like expressions (possibly with path references (=>)), function calls, or pairs of procedure calls (one for each direction). VML includes sophisticated collection management facilities. For example a *bijection* can be used to conditionally partition objects in an associated aggregate. In Figure 4 bijections partition objects in *space_face*'s *openings* (collection) attribute between the separate *windows*, and *doors* collections in *wall*. The mapping of each object involved in the bijection is via other *inter_class* specifications. Method calls can also be mapped, such as the mapping of the *space_face select* method to the *wall highlight* method. *Initialisers* specify default data or constructor calls for objects created when mapping. A programming environment (VPE) supports VML programming in both textual and abstract graphical form [3].

```
inter_class([space_face, material_face,
     building], [wall],
invariants(
    pace_face.location = 'ext',
    material_face.type_of_face = 'wall',
    plane_equivalence(space_face.plane,
                      material_face.plane),
     ),
equivalences(
     space_face.orientation =
wall.orientation,
     aterial_face.material=>r_value =
                      all.r_value,
     space_face.max=>x - space_face.min=>x =
                      wall.width,

bijection(space_face.openings[]@class(window),
                      wall.windows[]),
bijection(space_face.openings[]@class(door),
               wall.doors[]),
     space_face@select = wall@highlight
     ),
initialisers(
     wall.colour = 'light',
     wall@create(
         building.environment=>degree_days)
))
```

Figure 4. Example VML inter-class specification.

VML could be compiled to Snart constraints, but we have taken a more complex transaction-based approach. A mapping manager supports incremental or batch transfer of data between instances of the common repository and instances of tool models. Each attached view accumulates *transactions* or sets of modifications to be mapped to the common repository. On commital, the manager uses the *inter_class* specifications to determine what objects to create, delete, or modify in the common repository. The size of a transaction depends on user and tool needs. Large transactions equate to a batch version merging style. Small transactions, corresponding to individual tool manipulations, provide a more synchronous approach. Changes propagated to the common repository are made available to other views for incorporation via their mappings. The manager detects conflicts between changes, invoking negotiation processes to resolve them. The manager also maintains traceability, so that the originators or modifiers of data can be determined. This can be used in conflict arbitration (preferring user-supplied data over defaults, for example).

5. SERENDIPITY

To make effective use of multiuser/view environments, work process modelling and coordination mechanisms are needed. Serendipity is a multiview process modelling, enactment and work planning environment. It also supports event handling and group communication and awareness mechanisms [14].

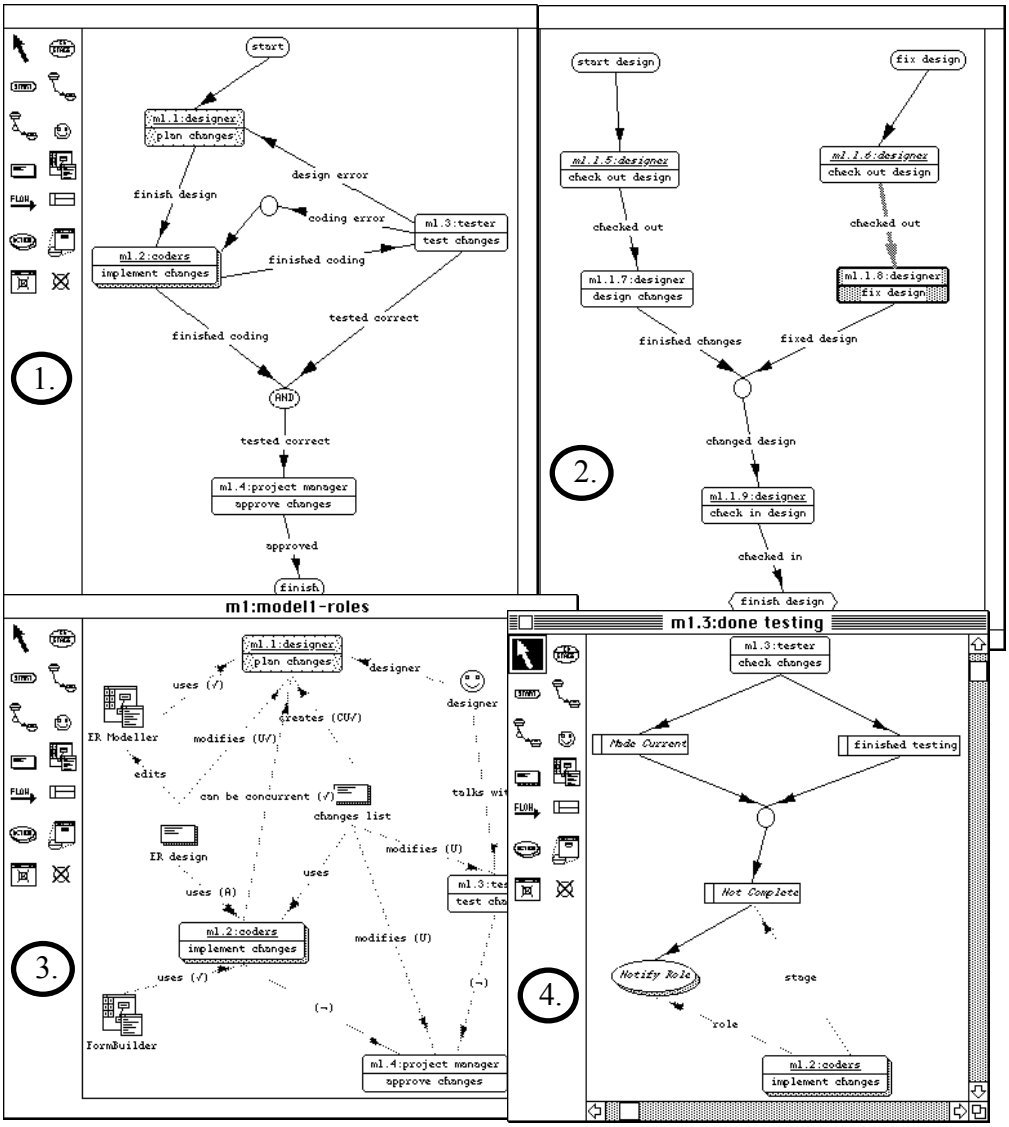


Figure 5. Software process model views and dialogs in Serendipity.

Figure 5 shows a process model for updating a software system (view 1). The notation adapts and extends Swenson's VPL [21], to support artefact, tool and role modelling, and event handling mechanisms. Stages describe steps in the process of modifying a software system. Enactment *event flows* link stages. If labelled, the label is the *finishing state* of the stage the flow is from. Shadowing indicates that multiple implementers can work on a stage (ie it has multiple subprocess enactments). Other items include *start*, *finish, AND*, and *OR* stages. Underlined stage IDs indicate a *subprocess* model, for example view 2 is a subprocess for view 1. Italicised stages indicate reuse of a *template* process model Serendipity also supports artefact, tool and role modelling, as in view 3, showing a different perspective of view 1. *Usage connections* indicate how stages, artefacts, tools and roles are used.

Enacted stages are highlighted by colour and shading. The shaded stage with a bold border ("m1.1.8:fix design") is the *current enacted stage* for the user i.e. their current work context. As a stage completes, event flows activate to enact linked stages. Enactments of stages are recorded, as are process model changes, and all enacted stages for a user can be shown in a "to-do" list dialog.

Filters and *actions* specify arbitrary enactment and work artefact modification event processing. View 4 shows an example of enactment event filtering. Filters "Made Current" and "finished testing" determine if "m1.3:check changes" has been made the current enacted stage or has been finished. If so, then if the "m1.2:implement changes" process has not completed (determined by filter "Not Complete"), the role associated with this stage is notified of testing being started or completed.

6. SUMMARY

Each of the preceeding approaches has advantages. MViews enables a range of consistency mechanisms that are difficult or impossible to achieve in our other approaches. Snart constraints are a language-based approach avoiding the need for procedural

code. However, structural mappings are difficult because Snart responds to data value changes. Thus it cannot easily deal with the consistency issues related to insertion and removal of objects in complex collections. VML is, however, well suited to handling such mappings. Serendipity is especially useful for event detection and response. Its use is currently restricted to work process modelling, but it will be productive to combine this approach with the propagation techniques of our other approaches.

All of the above approaches are implemented in an object-oriented extension of Prolog. This has enabled more rapid prototyping of systems than by using a conventional language. However, it has led to two problems: efficiency, due to a mismatch between the Prolog model and an imperative approach; and portability, due to the use of Prolog and high-level MacProlog-specific graphics. We have almost completed an extension to Java that incorporates Snart-style constraints. The extended language is currently compiled into Java source. With JIT compilation, we expect close to a 1000 times increase in constraint propagation speed over Snart. The next step is to extend the Java constraints with VML mappings, using compilation to avoid unnecessary mapping interpretation. To include the benefits of MViews, the low-level triggering mechanism that implements constraints will be extended to provide full information about changes so that change descriptions can be generated, where needed. Finally, Serendipity-style event-handling will be incorporated, permitting more direct event management. The unification of these mechanisms will allow us to select the appropriate means of achieving consistency between multiple views, from the declarative approach of constraints and mappings, to the lower-level but more powerful change description approach.

REFERENCES

1. Amor, R. and Hosking, J.G. Multi-disciplinary views for integrated and concurrent design. *International Journal of Construction Information Technology 2*, 1 (1994), 45-55.
2. Amor, R., Augenbroe, G., Hosking, J.G., Rombouts, W., and Grundy, J.C. Directions in modelling environments. *Automation in Construction* , 4 (1995), 173-187.
3. Amor, R.W., Hosking, J.G., and Mugridge, W.B. A declarative approach to inter-schema mappings. In *Modelling of Buildings Through Their Life-Cycle: Proc CIB W78/TG10 Conference,* CIB Proceedings Publication 180, Stanford, August 1995.
4. Augenbroe, G. A Joint European Project Towards Integrated Building Design Systems. *Building Systems Automation-Integration* (1993), 731-744.
5. Avrahami, G., Brooks, K.P., and Brown, M.H. A Two-View Approach to Constructing User Interfaces. *ACM Computer Graphics 23*, 3 (1990), 137-146.
6. Böhms, H.M. and Storer, G., "Architecture, methodology and Tools for computer integrated Large Scale engineering (ATLAS) - Methodology for Open Systems Integration," , no. ESPRIT 7280, 1993.
7. Freeman-Benson, B., "Constraint imperative programming," Technical Report, University of Washington, Seattle, 1991.
8. Grundy, J.C. and Hosking, J.G. A framework for building visusal programming environments. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, 1993, pp. 220-224.
9. Grundy, J.C. and Hosking, J.G., "Constructing Integrated Software Development Environments with Dependency Graphs," Working Paper, Department of Computer Science, University of Waikato, 1995.
10. Grundy, J.C. and Venable, J.R. Providing Integrated Support for Multiple Development Notations. In *Proceedings of CAiSE'95,* Finland, June 1995, LNCS 932, Springer-Verlag, pp. 255-268.
11. Grundy, J.C., Hosking, J.G., Fenwick, S., and Mugridge, W.B. Connecting the pieces, Chapter 11 in *Visual Object-Oriented Programming,* Burnett, M., Goldberg, A., Lewis, T. Eds, Manning/Prentice-Hall (1995).
12. Grundy, J.C., Hosking, J.G., and Mugridge, W.B. Supporting flexible consistency management via discrete change description propagation. to appear in *Software - Practice and Experience*.
13. Grundy, J.C. and Hosking, J.G. ViTABaL: A Visual Language Supporting Design By Tool Abstraction. In *Proceedings of the 1995 IEEE Symposium on Visual Languages,* IEEE CS Press, Darmsdart, Germany, September 1995, pp. 53-60.
14. Grundy, J.C., "Serendipity: integrated environment support for process modelling, enactment and improvement," Working Paper, Department of Computer Science, University of Waikato, 1996.
15. Hosking, J.G., Mugridge, W.B., and Blackmore, S. Objects and constraints: a constraint based approach to plan drawing. In *Proceedings of TOOLS 15,* Prentice-Hall, Sydney, 1994, pp. 9-19.
16. Krasner, G.E. and Pope, S.T. A Cookbook for Using the Model-View-Controller User Interface Paradigm in Smalltalk-80. *Journal of Object-Oriented Programming 1*, 3 (1988), 8-22.
17. Mugridge, W.B., Grundy, J.C., Hosking, J.G., and Amor, R., *Snart94 Reference/User Manual*, Department of Computer Science, University of Auckland, 1995.
18. Ratcliffe, M., Wang, C., Gautier, R.J., and Whittle, B.R. Dora - a structure oriented environment generator. *IEE Software Engineering Journal 7*, 3 (1992), 184-190.
19. Reiss, S.P. Connecting Tools Using Message Passing in the Field Environment. *IEEE Software 7*, 7, 57-66.
20. Reps, T. and Teitelbaum, T. Language Processing in Program Editors. *COMPUTER 20*, 11 (November 1987), 29-40.
21. Swenson, K.D. A Visual Language to Describe Collaborative Work. In *Proceedings of the 1993 IEEE Symposium on Visual Languages,* IEEE CS Press, 1993, pp. 298-303.
22. Venable, J.R. and Grundy, J.C. Integrating and Supporting Entity Relationship and Object Role Models. In *Proceedings of OO-ER'95,* Gold Coast, Australia, 1995, LNCS 1021, Springer-Verlag.