Using change descriptions to maintain consistency across multiple representations

John Hosking
Department of Computer Science
The University of Auckland
Auckland, New Zealand
john@cs.auckland.ac.nz

John Grundy
Department of Computer Science
The University of Waikato
Hamilton, New Zealand
jgrundy@waikato.ac.nz

Abstract

We describe a technique for dealing with partial mappings between different representations, both formal and informal, of an evolving software system. This technique uses discrete "change descriptions" to propagate changes between related views. These change descriptions my be used to autiomatically modify affected views, or to annotate the view to indicate manual intervention is required to maintain consistency.

Introduction

In developing software systems, it is common to make use of a mixture of informal and formal design notations, textual documentation, and target code, to assist in specifying, designing and implementing the systems. Each such notation has strengths and weaknesses. For example Entity Relationship diagrams provide a good way of describing system data and their relationships, while Data Flow Diagrams provide a useful way of expressing process and information flow. Together, the different views of the system that each notation represents assist in a holistic understanding of the needs, requirements and implementation of the system.

The problem with using such diverse notations, however, is the difficulty in maintaining consistency between each of the views of the system they represent. Of particular importance is the need to incrementally maintain consistency as the system evolves.

With purely formal notations, such as Object-Z [Duke et al, 1991], there exist refinement tools to help in transforming from the system description using the formal notation to an implementation in a programming language. More typically, however, direct or automatic translations between changes to views in one notation to corresponding views in another notation are difficult if not impossible. The reason for this are twofold:

- there is usually a creative element in refining from a more abstract to a less abstract notation (eg from a design level notation to code)
- elements in one notation may not have a direct representation in the other notation (eg processes in DFDs do not have a direct representation in ER diagrams) and either creative input may be required to perform the mapping or information may be lost on doing the mapping (eg between DFDs and ER diagrams).

In this paper we describe an approach that assists in maintaining consistency between views using different notations, even when creative input is required to completely implement a mapping. This approach is based on the propagation and presentation of discrete "change descriptions" between different views likely to be affected. We commence with a description of the MViews framework which supports change descriptions and their propagation. The use of change descriptions to either automatically modify affected views, or to annotate views with changes that require creative input by the programmer is then described. This is followed by a dicussion of presentation techniques for change descriptions in annotated views, and a description of extensions to the basic approach, followed by a summary and conclusions.

MViews

MViews [Grundy et al, 1993] is a framework for constructing integrated software development environments (ISDEs). The abstract syntax, semantic attribute values and multiple views of a software system are represented as graphs. Graph components are modified by operations to construct a program, and software developers view and manipulate the view graphs in concrete textual and graphical forms. MViews has been used to construct a number of IDSEs, most notably SPE, an environment for textual and graphical development of systems implemented in Snart, an object oriented Prolog [Grundy et al 1994] (Fig. 1).

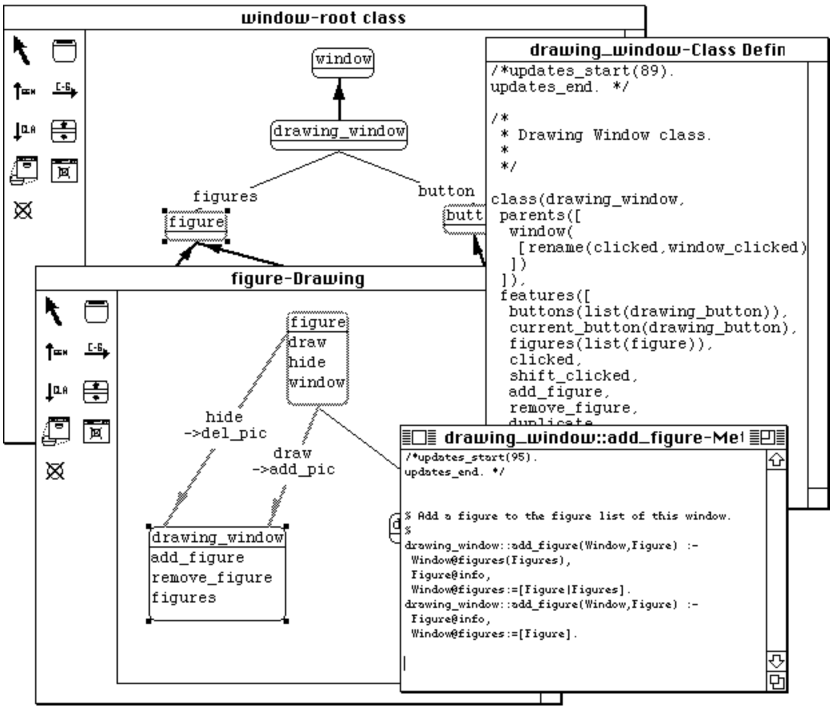

Fig. 1 A screen dump from the Snart Programming Environment (SPE).

MViews graphs are organised in a number of layers (Fig. 2). View layers are subgraphs representing part of a software system rendered in a particular notation. A shared base layer acts as a canonical representation of the software system with view relationships mapping between each view layer and the base layer.

Consistency management between updated graph components is supported by a novel change propagation mechanism. Graph operations generation discrete descriptions of the changes they make (called *change descriptions*). Once generated, change descriptions are broadcast to related (dependent) graph components (eg from a view layer component to its corresponding base layer component), which then respond to these update records and update their own state to maintain consistency.

Components receiving change descriptions are free to interpret them in any sensible fashion. Typical usages of relevance to this paper include:

1) To trigger a direct translation from a change in one representation to a change in another representation. For example, the addition of an attribute to a graphical OOA-like view in SPE causes the addition of a feature definition in a Snart code implementation view. This type of direct translation is usually appropriate for straightforward translations between "close" representations.

2) Where a choice of translations is possible, the user can be presented with the range and asked to select one. This is similar to what is done in many refinement support tools [eg Groves and Nickson, 1988].

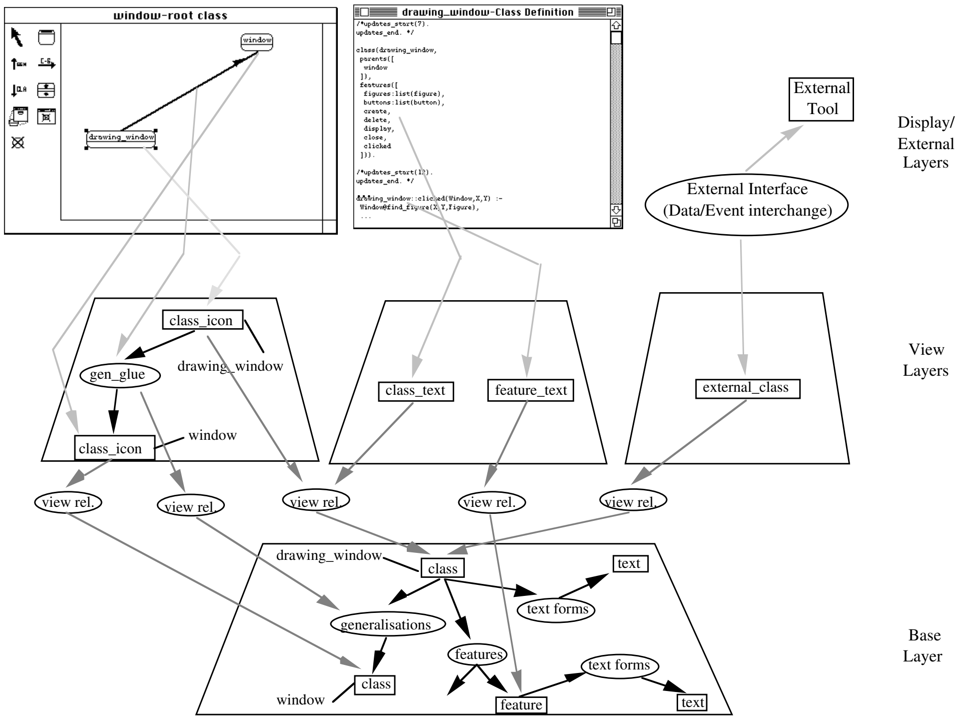

Fig. 1. Multiple views of an SPE program in MViews.

3) To annotate a view. Where a direct translation is impossible, the receiving view can simply insert an indication into the view that a change has occured that potentially requires manual intervention to make things consistent. For example, in SPE the addition of an abstract relationship in an OOA view causes the insertion of a textual annotation describing the change in the corresponding Snart implementation views. The programmer then creatively decides the appropriate implementation (eg perhaps as a method call or as an object reference).

The latter case is of particular interest. The annotation technique does not automatically ensure consistency between views. Rather it supports a form of partial inconsistency where each view knows about, and has documented changes to the other views that can potentially cause inconsistency.

As an aside, partial inconsistency between views is occasionally useful, even when direct translations are possible. In SPE, for example, all changes affecting code views are initially presented as annotations. The programmer may review each of the changes prior to having the system automatically effect them.

While change description annotations are obviously useful, they raise a number of presentation issues including how and when to annotate. These issues are discussed in more detail in the following sections.

Presentation of change descriptions in annotated views

One useful annotation technique for textual views is to "unparse" the change description into a readable textual form and insert this as a comment into the textual view. As mentioned in the previous section, this technique has been used in SPE code views. Change descriptions relating to a class are inserted as a comment into the class header. An example is given in Fig. 3. A change has been made in a graphical design view: a client-supplier call has been added from the *clicked* method of the *drawing_window* class to the *pt_in_figure* method of class *figure*. Neither the place in the method the call is made nor the arguments to the call were specified in the graphical view. The user thus must insert these manually into the textual view to make the design and code views consistent.

This unparsing approach is useful for all types of textual view. For formal views, such as a (textual) Object-Z view, change descriptions documenting changes to informal specifications can be inserted into the formal view, and vice versa. For informal views, such as a documentation view for a class, the annotation can be used to automatically insert a history of changes made to that class, or to selectively update key words in the text. This technique has been used in SPE to create an editable modification history, supplemented by user generated change descriptions providing a higher level description of a sequence of low level changes (Fig. 4).

```
/* updates start(94)
update(36). % add client/supplier design call:
            clicked -> figure::pt_in_figure
updates_end. */

class(drawing_window,
  parents([window([rename(clicked,window_clicked)])
         ]),
  features([
     buttons:list(drawing_button),
     clicked,
      ....
]))).
```

Fig.3 Unparsed change description in SPE textual view

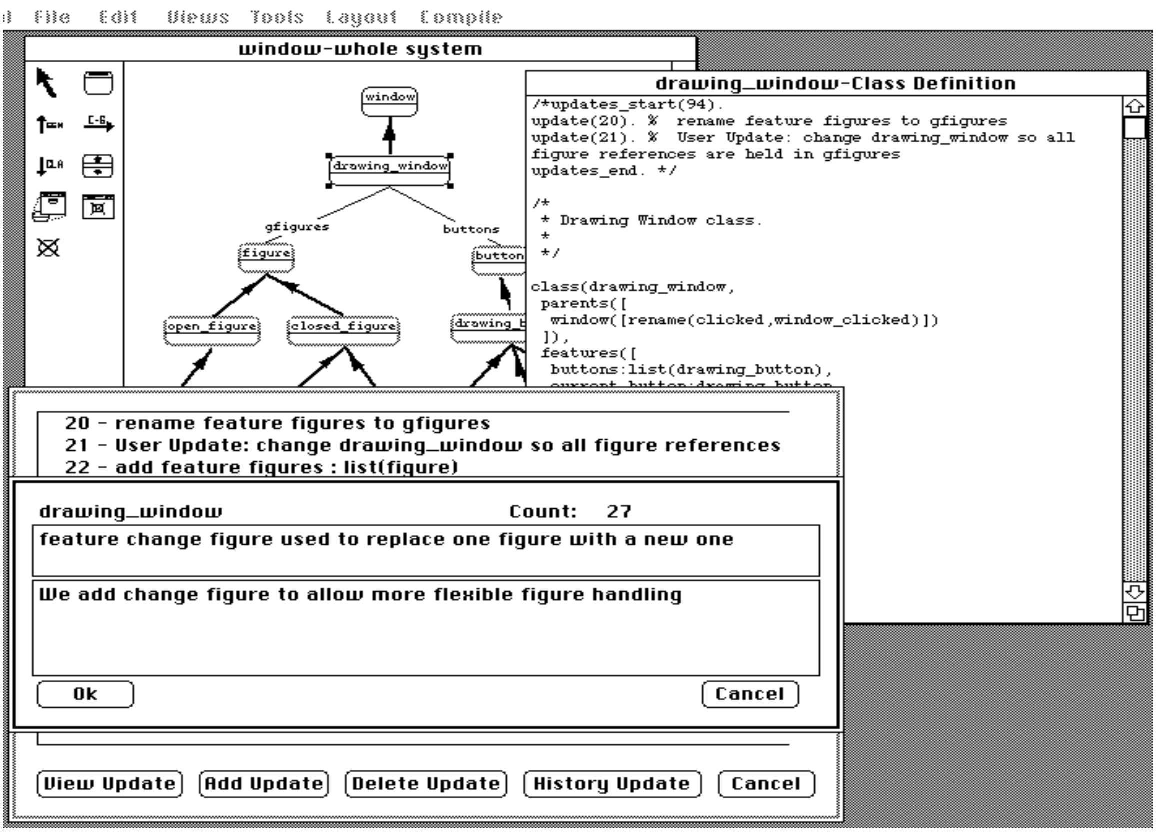

Fig. 4 Editable change history in SPE.

The textual unparsing approach can also be used to effect to annotate diagrammatic (graphical) views. Changes made in other views can be presented in a scrollable dialogue for review and incorporation into the graphical view. This technique has been used to good effect in an experimental extension to SPE supporting collaborative interaction [Grundy et al, 1995].

A wide range of iconic annotations can be also be made in graphical views. An icon can be marked in some way, eg colour, to indicate that there is an associated change description generated by another view. For example, SPE greys icons to indicate the component they represent has been deleted as a result of modifications to another view. The annotated icon can then be selected and the change description rendered textually or used to generate a hypertext link to the view generating the original change (with that change possibly highlighted).

Other possibilities include the addition of extra icons, similar to copy editor's marks, giving an indication of the type of change that has occured - a form of graphical "unparsing" of the change descriptions.

Extensions of approach

The basic change description annotation mechanism can be extended to provide higher levels of support. Here we describe two such extensions which we are currently experimenting with.

The first extension is to permit the user to associate a sequence of changes plus selections from existing code with a change description generated from another view. For example, in Fig. 1, the actual code in the clicked method implementing the call to *figure::pt_in_figure* could be associated with the change description in the class header. This association could then be used in a number of ways such as:

- as a hypertext link from the design view link to the code implementing that design element (clicking appropriately on the design element could bring up the code view with the implementation highlighted). This link can thus act as an aid in tracing from analysis requirments through design decisions into implementation code and as a tool to assist with impact analysis.

- to annotate the design element with a change description if changes are made to any part of the associated implementation. This would be used to indicate that the change may have resulted in a violation of the intent of the design element. This is a much more integrated and incremental approach than reverse-engineering of design diagrams.

The second, somewhat related, extension is to allow a sequence of low level changes to be considered to be a "transaction" with an associated user generated change description that describes the rationale behind the transaction as a whole (an extension of the user generated change decscriptions of Fig. 4). This allows a more coherent approach to translating between representations, where groups of changes, some with possible automatic translations and some without, can be considered together. Additional presentation techniques, such as a playback of changes, are useful to support this extension and have been experimented with in [Grundy et al, 1995].

Summary and conclusions

We have argued that mixing a variety of representations, formal and informal, is both useful in developing software, and problematic because of the resulting consistency problem. A partial solution to this is to provide autmatic translations between the representations, where possible, but to also use a combination of view annotations and manual modification where this is impossible. extensions of this basic approach include support for tracability and impact analysis.

The MViews ISDE framework with its change description propagation mechanism has proved a useful vehicle for experimenting with these ideas.

References

Duke, R., King, P., Rose, G., and Smith, G., The Object-Z specification language: Version 1. Technical report 91-1, Software Verification Research Centre, Department of Computer Science, University of Queensland, May, 1991.

Groves, L and Nickson, R., An intelligent editor for constructing correct programs. in Proc 3rd New Zealand Conference on Expert Systems, NZESIG, 1988.

Grundy, J.C. and Hosking, J.G.: Constructing multi-view editing environments using MViews, *Proc. 1993 IEEE Symposium on Visual Languages*, IEEE Computer Society Press, Los Alamitos, CA, 220-224,1993.

Grundy, J., Hosking, J., Fenwick, S., and Mugridge, W.: Connecting the pieces, to be published as Chapter 11 of *Visual Object-Oriented Programming*, Margaret Burnett, Adele Golberg, and Ted Lewis (eds), Manning Publications, 1994.

Grundy, J., Mugridge, W.B. Hosking, J.G. and Amor, R.A., Support for collaborative, integrated software development, accepted for publication in *Proc 7th Conference on Software Engineering Environments*, to be held in The Netherlands, April 1995.