AUTOMATED ENHANCEMENT OF KNOWLEDGE REPRESENTATIONS[1]

Robert Balzer
USC/Information Sciences Institute
4676 Admiralty Way
Marina del Rey, CA 90292

ABSTRACT

A new class of maintenance tool is presented, designed specifically for enhancement of knowledge representation systems. These tools are based on the structure of a domain model and the ways it can be changed. A complete language for changing domain models is presented. Associated with each change is an explicit assumption in the previous model that is violated. A complete analysis is presented of how existing knowledge representation frames and the associated operations on those frames must be updated to correct these assumption violations introduced by changing the domain model.

Such tools are an important first step towards support for an incremental development process. Almost no support currently exists for enhancing knowledge representation systems. The tools described here support enhancements involving modification to the domain model. Support for the remaining enhancements will require both more knowledgeable tools which understand the system they are manipulating, and more declarative structural specifications for such systems which facilitate that understanding.

INTRODUCTION

It is by now well known that maintenance is the dominant life-cycle cost for conventional software systems, often consuming 80-90% of the total effort. Yet this life-cycle phase has received remarkably little attention or support. The few tools that exist (such as debuggers, modification audit trails, configuration managers and regression testers) have been available for a long ume and are quite stable. There is scant evidence of further progress.

In fact, the major thrust of the last several years in the Software Engineering community has been towards reducing the need for maintenance, rather than facilitating it. This thrust has centered on reducing (or eventually eliminating) implementation bugs through improved development methods (either informal or formal and either manual or automated). This thrust has begun to bear fruit. Improvements have been made in reducing the number of implementation errors. They may even be eliminated entirely via formalization and automation of the development process.

But this thrust begs the maintenance problem because it only addresses one maintenance activity, the correction of implemenution bugs. This activity decays rapidly after a system is fielded, and the thrust merely lowers the initial level of that activity.

Enhancement, rather than correction of implementation bugs, is the dominant maintenance activity. It grows rather than decays over time. It is the source of the multi-year maintenance backlog experienced by most DP organizations. It will become even more dominant as the development improvement thrust partly reduces the need to correct implemenution bugs, and as the current trend towards release of systems in incremental stages, with enhanced functionality, accelerates.

There are two reasons for such enhancements. The first is that no-one has enough insight to build a system correctly the first ume (even assuming no implementation bugs). The second is that the mere existence of the system, and the insight gained from its usage, create a demand for new or altered facilities.

Rather than attempting to eliminate the need for maintenance we should recognize that *enhancement,* not initial development, *is the central software activity* (and the basis for achieving the "softness" promised in software).

This dominance of enhancement is even greater for Knowledge Based systems because they are developed via cut-and-try explority programming techniques. Our software life-cycle and support environments should be rethought accordingly.

We have elsewhere presented our knowledge based version of such a paradigm and support environment [Balzer et al 83a]. This paper addresses the enhancement activity directly. It defines two types of enhancements, structural and functional, and focuses on the former. It categorizes the class of possible structural enhancements, defines a complete language for specifying such enhancements, and describes an implemented set of tools for effecting these structured enhancements. It also presents an analysis of the changes to existing knowledge representauon frames and the operations on them necessitated by these structural enhancements.

The basis for all these capabilities is an explicit, modifiable domain model which defines the class of valid frames. The structure of this model determines the class of assumptions that can be explicitly represented. Change to such an assumption should cause propagation of effects to all places in the frames and the operations on them where that assumption was relied upon. Our leverage arises from the fact that the domain model forces these assumptions to be explicitly stated, and to be stated in such a way that reliance can be mechanically determined. This provides the basis for a new class of maintenance tools which support the propagation of effects into the frames and operations associated with a domain model.

THE STRUCTURE OF A DOMAIN MODEL

We therefore begin by examining the structure of a domain model. We have a fairly conventional frame-based object model [Minsky

This research is supported by the Defense Advanced Research Project Agency under Contract No MDA903 83 C 0335 Views and Conclusions contained in this report are the authors and should not be interpreted as representing the official opinion or policy of DARPA. the US Government, or am person or agency connected with them

81, Roberts 77, Bobrow 76] in which the data is self-describing (i.e., its type(s) can be determined), all instances of a type can be obtained, attribute declarations are inherited along subtype links, the range of an attribute is type limited, and an attribute can be either required or optional and either single or multi-valued. Our knowledge representation language is quite a bit more elaborate [Balzer et al 81] but only these aspects are relevant to maintenance.

STRUCTURAL ENHANCEMENT

The set of all possible enhancements can be divided into two categones - those that change the domain model and those that do not. We call the former structural enhancements, and the latter functional enhancements. The types of structural enhancement possible are determined by the structure of the domain model. This leads fairly directly to a complete language for stating such modifications. More importantly, the explicit declarative nature of the model makes it clear what assumption is being violated by each structural enhancement. This knowledge can then be used to identify all the sites in the frames and operations which relied upon the assumption. The complete edmng language and the ability to identify all sites within the frames and operations that must be updated provides the basis for the new class of maintenance tools presented here, and the rauonale for focusing exclusively on structural enhancement.

There are two basic types of structural enhancement: changing the attribute structure and changing the type structure. The domain and range of an attribute can be generalized (picking a supertype of the current type), specialized (picking a subtype of the current type), or changed incompatibly (picking a type which is neither a supertype nor a subtype of the current type). The cardinality restriction can be changed from one value in the enumerated range -(UNIQUE, OPTIONAL, MULTIPLE, ANY} to another.

Changes to the type structure involve changing the membership of some enumerated set. The kinds of changes possible are simply adding an item to or deleting an item from the enumerated set, refining an existing item into several new ones, or combining existing items into a new abstraction. These changes are handled analogously whether the "enumerated set" is the set of instances of a type, the subtypes of a type, or the attributes defined on a type.

AN EXAMPLE STRUCTURAL ENHANCEMENT

In order to illustrate the utility of structural enhancements and the modifications involved to effect such changes, we have selected a single actual example which we will use throughout this paper. This example was completely effected by the tools described here with the exception of propagation into the operations which are not yet implemented. Here we will describe the original domain model, the change we wish to make, and the modifications to the model required to effect that change. Later sections will return to this example to describe the propagation of these effects into the existing frames and operations.

The example we've chosen arose in our work on developing an automated environment for software development [Balzer et al 83b]. As part of that effort we needed to represent the objects in the programming domain. A piece of this domain model is shown graphically in the top portion of Figure 1. Circles (or ovals) are used to represent types. The subtypes of a type are placed below it and connected to it by unlabeled links. Boxes are used to indicate attributes associated with the type. We have suppressed showing the range and count specification of the attribute to avoid cluttering the diagram.

This diagram indicates that the type PROGRAM-OBJECT has FUNCTION, VARIABLE. RECORD, etc.. as subtypes, and COMPONENT. SOURCE-TEXT, and MAINTAINER, as attributes. This type was being used to represent the individual program objects such as functions, variables, records, etc., which had a SOURCE-TEXT that defined them, as well as the hierarchical module structure. Modules had COMPONENTS which were either other modules or individual program objects. The PROGRAM-OBJECT type was clearly being overloaded, representing semantically different objects based on whether they had components or a source-text. We decided to make this distinction explicit, as shown in the bottom diagram of Figure 1, b\ refining the PROGRAM-OBJECT type into INDIVIDUAL-PROGRAM-OBJECT, and MODULE subtypes which respectively had SOURCE-TEXT and COMPONENT attributes. Furthermore, the existing subtypes of PROG RAM-OBJECT should now become subtypes of INDIVIDUAL-PROGRAM-OBJECT.

All of these changes are accomplished by a single use of the REFINE-TYPE tool. The programmer indicates what type is to be refined (PROGRAM-OBJECT), what new types (INDIVIDUAL-PROGRAM-OBJECT and MODULE) are to be created as subtypes of it. which attributes of the refined type are to be specialized to these new subtype (SOURCE-TEXT to INDIVIDUAL-PROGRAM-OBJECT and COMPONENT to MODULE), and which existing subtypes should become subtypes of the new subtypes (FUNCTION, VARIABLE, RECORD, etc.. all become subtypes of INDIVIDUAL-PROGRAM-OBJECT). The REFINE-TYPE tool invokes other tools to accomplish these modifications to the domain model. For instance, after the new subtypes have been created, attributes are moved to them from the supertype via the SPECIALIZE-ATTRIBUTE-DOMAIN tool.

One change remains. We wish to make the two moved attributes REQUIRED instead of OPTIONAL. This requires two invocations of the CHANGE-CARDINALITY-RESTRICTION tool.

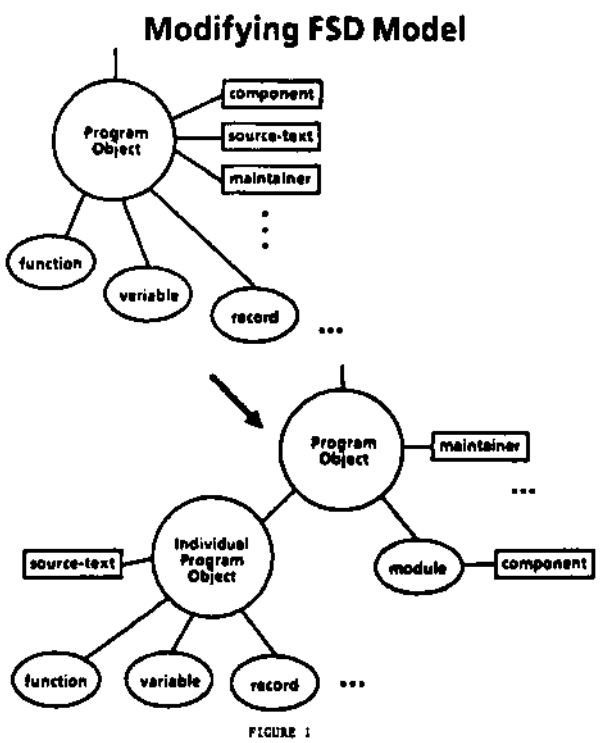

FIGURE 1

The effects of these changes to the domain model on the existing frames and operations will be described in the section dealing with those topics.

PROPAGATING STRUCTURAL ENHANCEMENTS INTO EXISTING FRAMES

Having a complete language for stating structural enhancements, and editing tools for making the necessary modifications to the domain model is a nice start. But it is only a beginning towards the goal of maintaining a system through such structural enhancements. Having altered the domain model, the effects of those changes must be propagated through both the operations that use the model and the frames already created by (or for) those operations.

This propagation of effects is possible because the explicit declarative nature of the domain model forces the structural assumptions to be explicit. This enables us to identify which ones are violated by the enhancement. The structural nature of these assumptions allows us to detect all sites within the frames and operations which rely upon each assumption. This ability to identify the "usage" sites is the basis for the new maintenance tools presented below. This section focuses on propagating the effects of a structural enhancement into existing frames: the next section addresses propagation into operations. These propagation effects are summarized in Figure 2.

Cardinality Restriction Modification

When the cardinality restriction of an attribute is changed, only two assumptions can be violated. The first is the assumption that the attribute is required. If an optional attribute becomes required then instances without the attribute must have it added. If a required attribute becomes optional, then the existing frames are still valid, but the programmer may wish to remove some instances of the attribute. The second assumption that a cardinality restriction change can violate is that an attribute is single-valued. If a multi-valued attribute becomes single-valued, then instances with more than one value must be paired down to a single value. If a single-valued attribute becomes mulu-valued, then the existing frames are valid but the programmer may wish to augment existing instances.

Attribute Range Modification

When the range of an attribute is changed, then only the assumption about the type of the value of that attribute is affected. If the range is generalized, then the existing data is valid, but the programmer may wish to generalize the value of some existing instances. If the range is specialized, then some instances (those whose attribute value is outside the specialization) are invalid and their attribute value must be replaced (if the attribute is not required, it can be deleted rather than replaced). If the range is changed incompatibly, then all existing instances are invalid and their attribute value must be replaced (or alternatively, deleted if the the attribute is optional).

Attribute Domain Modification

When the domain of an attribute is changed, then only the assumption of which type of object has this attribute is affected. If the domain is generalized, then the attribute is applicable to more instances. If it is required, then it must be added to the newly applicable instances. If it is optional, then the existing frames are valid, but the programmer may wish to add it to the newly applicable

Model Modification Summary

MODEL MODIFICATION	KNOWLEDGE BASE UPDATES	OPERATION UPDATES
Count Spec Changes		
Optional ==> Required	Augment instances without newly required attribute	Deconditionalize consumer access Ensure creator specifies attribute
Required ==> Optional	Existing frames are valid Attribute can be eliminated from existing instances	Conditionalize consumer access
Single-valued ==> Multi-valued	Existing frames are valid Instances can be augmented by additional values	Introduce loop or non-deterministic access for producer/consumer access
Multi-valued ==> Single-valued	Retain only single value	Eliminate loop or non-deterministic access
Attribute Range Changes		
Generalize Range	Existing frames are valid Range can be generalized	Conditionalize consumer access
Specialize Range	Replace or delete overly general attribute	Remove eliminated cases from conditionalized consumer access Ensure producer only generates subtype
Incompatible Range	Replace or delete invalid attribute values	Check producer/consumer type assumptions
Attribute Domain Changes (Attribute Movement)		
Generalize Domain	(Possibly) augment supertype instances with attribute	Optionally introduce attribute processing into supertype handling
Specialize Domain	Remove attribute from instances outside new domain	Conditionalize producer and consumer access
Incompatible Domain	Move existing attribute values to corresponding instance of new domain	Introduce indirect access to corresponding instance of new domain via path from old domain
Set Membership Changes (Enumerated Ranges, Subtypes, and Attributes)		
Item Addition	Existing frames are valid New item can replace use of existing items	Add item to consumer case analysis
Item Deletion	Remove use of deleted item	Remove eliminated case from conditionalized consumer access Ensure producer doesn't generate deleted item
Item Refinement	Existing frames are valid New refinements can replace use of refined item	New refinements can replace use of refined item
Item Combination	Existing frames are valid New combination can replace use of combined items	New combination can replace use of combined items

FIGURE 2

instances. If the domain is specialized, then some instances (those which are not instances of the specialization) must have the attribute removed.

The case of an incompatible change in the domain of an attribute is most interesting. It could simply be handled as the deletion of the attribute from the original domain and the addition of it to the new domain. This would destroy the attribute value from instances of the original domain and require the programmer to specify a value (or values) for instances of the new domain. However, we feel that the intent of such a change is almost always to MOVE the existing attribute values from instances of the original domain to "corresponding" instances of the new domain. This correspondence is defined by a path through the knowledge base which maps an instance of the original domain mto an instance (or possibly several) of the new domain. The attribute values are moved as defined by this mapping. This path can either be specified by the programmer or heuristically determined (via the shortest path of required single-valued attributes).

Set Membership Modification

When the membership of a set changes, then only the assumption of the set of alternative values (the pigeon-hole principle) is affected. If an item is added to the set, then the existing frames are valid (attributes are added with OPTIONAL as the cardinality restriction), but the programmer may wish to use the new item in place of. or in addition to, existing items. If an item is deleted, then use of that deleted item must be removed or replaced. If an item is refined, then the existing frames are valid, but the programmer may wish to substitute one of the refinements for use of the refined item (i.e., be more specific). If items are combined to form a new abstraction, then the existing frames are valid, but the programmer may wish to substitute the new item for some uses of the combined items (i.e., be more abstract).

Example Frames Propagation

Continuing with the example structural enhancement shown in Figure 1 which refined the type PROGRAM-OBJECT into INDIVIDUAL-PROGRAM-OBJECT and MODULE, and specialized some of its attributes, we consider here the effect of these changes on the existing frames. First, the existing frames remain valid when a type is refined, but the programmer may wish to reclassify some instances of the refined types as instances of one of the refinements. In this example, we wish to partition the existing instances among the two subtypes (those that have a SOURCE-TEXT are to become INDIVIDUAL-PROGRAM-OBJECTs and those that have COMPONENTS are to become MODULEs). The REFINE-TYPE tool allows such predicates to be specified and performs the indicated reclassifications.

Second, the specialization of the domain of the two attributes may necessitate removal of this attribute from instances outside the specialized domain. In this example, since the existence of the attribute determined the reclassification, there are no such instances to update. However, the tool is unable to infer this result Instead, it dutifully checks for any such instances.

Finally, the changing of the cardinality restriction of these attributes from OPTIONAL to REQUIRED causes the system to check for instances that need, but do not already have, the attribute. Again, this search is fruitless because the reclassification was based on the prior existence of the attributes. All of these changes to the existing frames, as well was the searches for frames requiring and/or desiring change, were performed by the frame propagation portion of the enhancement tools.

PROPAGATING STRUCTURAL ENHANCEMENTS INTO OPERATIONS

The previous section identified the assumptions that were violated by each type of structural enhancement and described how dependencies upon those assumptions could be detected and corrected in existing frames. The analysis of the deduction of on those same assumptions and their correction in existing operations is very similar. However, since this portion of the analysis has not yet been implemented, only a summary of the analysis (as shown in Figure 2) is presented here. The nature of these corrections is typically to change the conditionally of the code which uses a changed portion of the domain model so that it agrees with the conditionality defined by the model.

The summary in Figure 2 distinguishes three types of usage of an attribute: consumer, producer, and creator. Consumer uses are all accesses to the value of an attribute; producer uses are all places which set, or remove, the value of the attribute, and, creator uses are the subset of producer uses which set the value of the attribute while creating an instance of the attribute's domain.

Example Operation Propagation

Returning once again to the structural enhancement example shown in Figure 1 which refined the type PROGRAM-OBJECT mto INDIVIDUAL-PROGRAM-OBJECT and MODULE, and specialed some of its attributes, we now consider what effects these changes have on the existing operauons.

First, when a type is refined, the existing operations remain \alid but the programmer may wish to replace some uses of the refined item by one of its refinements. In this case, basically all uses of PROGRAM-OBJECT were replaced by uses of one of the refinements. In those places where it was not known which refinement was present, a TYPECASE statement was inserted to make the selection.

Second, the specialization of the domain of the two attributes causes both the the producer and consumer uses to be conditionalized. For this example, this conditionalization (after simplification) is a NO-OP because all uses already occur inside of the proper subtype determinauon (i.e. the necessary conditionality already exists). This results from the fact that in this example the attribute occurrence corresponds exactly to the subtype definition.

Finally, the changing of the cardinality restriction of these attributes from OPTIONAL to REQUIRED causes the consumer uses (which remain valid) to be checked for unneeded conditional guards, and creates uses to include this attribute.

Unfortunately, since none of the tools for propagating effects into operations is yet implemented, all of these checks and modifications were performed manually.

CURRENT STATUS

Our goal in undertaking this effort was to obtain assistance in enhancing systems. Our investigation of enhancement led to the complete categorization of structural enhancement of (a subset of) our domain modeling language and the analysis of the propagation effects of such enhancements on both frames and operauons reported here. They, in turn, form the basis for a new class of tools for performing such enhancements. Implementation of these tools is well underway and consists of three phases. The first consists of tools for modifying the domain model as defined by the complete categorization presented earlier. These tools are implemented and are being used to provide structural enhancements to domain models.

The second phase consists of tools for propagating the effects of structural enhancements into existing frames. These tools are being implemented as they are needed. One of these tools, refining an item, has been used extensively (including the example provided in this paper) to apportion the existing instances of a refined type among its subtypes. Since many such instances may need to be reapportioned, the tool uses a programmer-supplied predicate for apportionment rather than interactively querying the programmer. The built-in facility for our system to access all instances of a type [Goldman 83] makes creating such tools straightforward. We anucipate no problems with the remaining tools in this phase.

The final phase consists of the tools for propagating the effects of structured enhancements into operations. Implementation is awaiting completion of both a static analyzer capable of detecting consumer, producer and creator uses of the types, attributes, and enumerated ranges from which our domain model is composed, and a type checker capable of determining the type of each value being produced and/or consumed by the operations. Once the static analyzer and type checker are available, we anucipate moderate difficulty in implementing these tools because of the syntactic and semantic variability allowed and because no-one has yet designed a suitable transformation system (as opposed to a set of ad hoc procedural manipulations) for making these kinds of modifications to operations. We believe that an important precursor to such a facility is a categorization of the types of program modification to be made, rules of composition, and rules for simplification. Such a foundation always appears to be necessary for formal manipulation of programs.

CONCLUSION

This paper addresses the dominant maintenance activity, enhancement. It focuses on those enhancements, called structural, that change the domain model. It shows how the domain model structures the validity assumptions of the domain and forces them to be explicitly stated. The structure of the model determines the type of changes that are possible to the model. These changes have been categorized into a complete language for specifying structural enhancements. Only four types of modification are possible: changing the domain, range, or cardinality restriction of an attribute, or modifying the set of items in an enumerated set (such enumerated sets include the type hierarchy, the set of attributes associated with a type, and explicitly enumerated ranges).

Associated with each of these changes is a single validity assumption affected by the change (in the case of a cardinality restriction change there are two). The fact that reliance on these assumptions can be detected in both frames and operations provides the basis for automated tools that propagate the effects of a structural enhancement into both the frames and operations associated with the enhanced domain model. A complete analysis of these propagation effects was presented for each of the four types of structural enhancement. Generally the propagation effects on the frames directly mirror the structural changes to the model, while those on the operauons are evidenced in the addition or removal of conditional guards on the producer and consumer uses of the modified structure.

One particularly interesting structural enhancement is an incompatible change to the domain of an attribute (as opposed to a generalization or specialization of the domain). Such a change is treated as a form of indirection and a mapping is used in both the frames and operauons to locate the corresponding object specified by the indirection. This mapping can either be specified as pan of the enhancement, or heuristically determined.

Automated tools for the complete set of domain model modifications and for the propagation of effects of some of these modifications into the existing frames have been built and are being used.

Such tools are an important first step towards support for an incremental development process [Balzer et al 83a] for both conventional and knowledge based software.

The tools described in this paper support structural enhancements. The remaining, functional, enhancements will be more difficult to support because they involve changing the procedural component, the operations, rather than the declarative and highly structured domain model It is our belief that such tools must understand the functional aspects of the system they are manipulating, and that this will require a more declarative structural specification for these systems.

REFERENCES

[Balzer et al 81] Balzer, R., N. Goldman, D. Wile, *Operational Specification as the Basis for Rapid Prototyping.* ACM Sigsoft Software Engineering Syposium onf Rapid Prototyping. Technical Report, October 1981.

[Balzer et al 83a] Balzer, R., C. Green, T. Cheatham. "Software Technology in 1990's," *Computer Magazine,* November 1983.

[Balzer et al 83b] Balzer, R., D. Dyer, M. Morgenstern, R. Neches, "Specification-Based Computing Environment," in *AAAI-83,* AAA1,1983.

[Bobrow 76] Bobrow, D.. and T. Winograd, *An Overview of KRL, A Knowledge Representation Language,* Xerox, Technical Report CSL-76-4, July 1976.

[Goldman 83] Goldman, Neil M., *APS Reference Manual* USC Information Sciences Institute, 1983.

[Minsky 81] Minsky, M., "A Framework for Representing Knowledge," in J. Haugeland (ed.), *Mind Design,* pp. 95-128, MIT Press, Cambridge, Mass. 1981.

[Roberts 77] Roberts, R., and 1. Goldstein, *FRL Users' Manual* Massachusetts Institute of Technology. Technical Report Al Memo 408,1977.