Modeling multiple views of common features in software reengineering for reuse

Stan Jarzabek and Chew Lim Tan

Department of Information Systems and Computer Science
National University of Singapore
Singapore 0511
stan@iscs.nus.sg, tancl@iscs.nus.sg

Abstract. Common objectives of software reengineering are to improve program maintainability, to port programs into new platforms or to support new functions. To meet reengineering objectives, sometimes it is necessary to substantially re-deign programs; then, reengineering becomes an opportune moment to address reusability. In the "reengineering for reuse" scenario, a reusability framework is built prior to reengineering efforts. Within the framework, potentially reusable features are modeled and representation structures for capturing reusable features are built. The core of the framework is a family of domain models. Domain models are built in the course of both reverse engineering of existing programs and independent domain analysis. Domain models consist of documentation templates, organized in Object-Oriented way, that describe common (therefore reusable) features and their implementation. Often we find that, apart from similarities, there are also some variations in feature specifications and implementation from one system to another. Modeling reusable features and capturing variations in feature specification is the topic of this paper.

1. Introduction

Recent surveys [20] show that investments in Information Technology (IT) do not yield expected benefits. Many of the aging business programs are expensive to maintain, run on outdated platforms and do not meet requirements of strategic information systems companies need today. In short-term, some of those programs must be restructured for better maintainability and converted into new computers, databases, operating systems, languages, etc. In long-term, however, programs must be reengineered (or re-written) to fully exploit advantages of new technology and to be in tune with company's strategic plans [16,21]. We call this strategic reengineering [13]. Strategic reengineering may involve re-designing program architecture or even change of the implementation technique (e.g., taking programs under control of a CASE tool or re-designing procedural programs into the Object-Oriented architecture). Strategic reengineering is expensive and must be cost-justified. Addressing reusability during reengineering can increase the value of a reengineering solution [14].

To ensure consistency of reengineering efforts with company's business and IT strategies, we defined a lifecycle model whose phase and process structure is shown on figures 1-3.

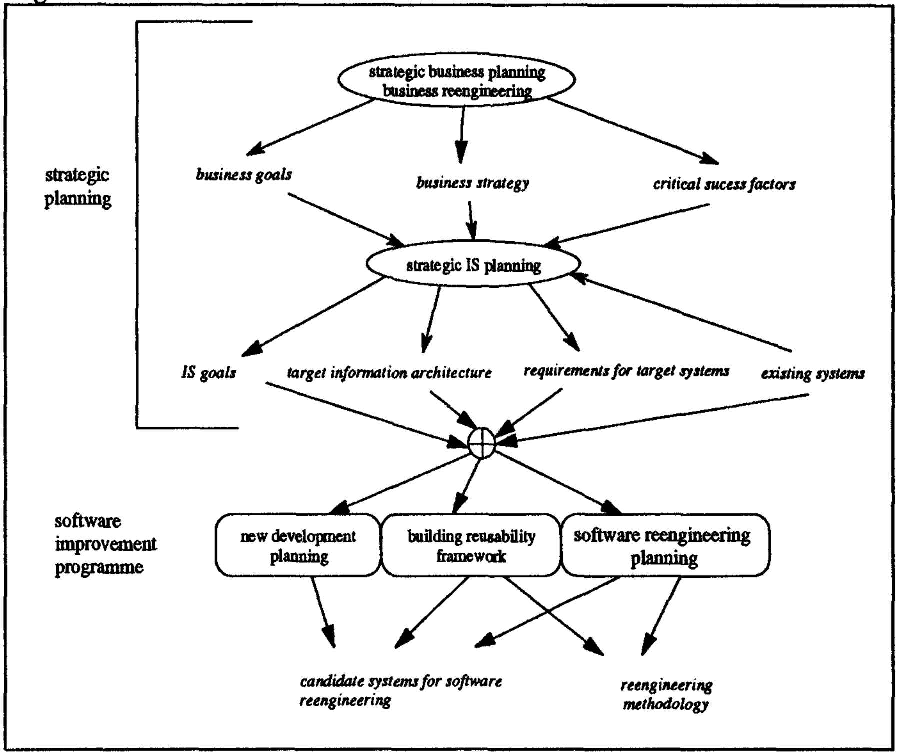
strategic business planning
business reengineering
business goals
business strategy
critical sucess factors
strategic planning
strategic IS planning
IS goals
target information architecture
requirements for target systems
existing systems
software improvement programme
new development planning
building reusability framework
software reengineering planning
candidate systems for software reengineering
reengineering methodology

Fig. 1 Strategic reengineering: lifecycle

Fig. 1 depicts reengineering as part of an overall software improvement program determined during strategic planning [16]. During business planning, a company clarifies business goals and modifies business operations to meet new goals and to take advantage of new IT options. Information System (IS) planning leads to identifying software systems a company needs in order to follow its business plan. A stable target computer/software architecture for future software development and maintenance activities is also defined. Company's existing platforms and programs are assessed and based on this assessment future development and reengineering efforts are planned.

The logical structure of the reengineering for reusability process is depicted on fig. 2. During reengineering, we transform an old system S into a new system, S-NEW. To facilitate program transformation, we recreate program views at various levels of abstraction. The physical level is created using reverse engineering techniques. The physical layer includes abstract syntax trees and design abstractions such as control flow and data flow graphs, procedure calling trees, data structure charts, various cross-reference lists, etc.

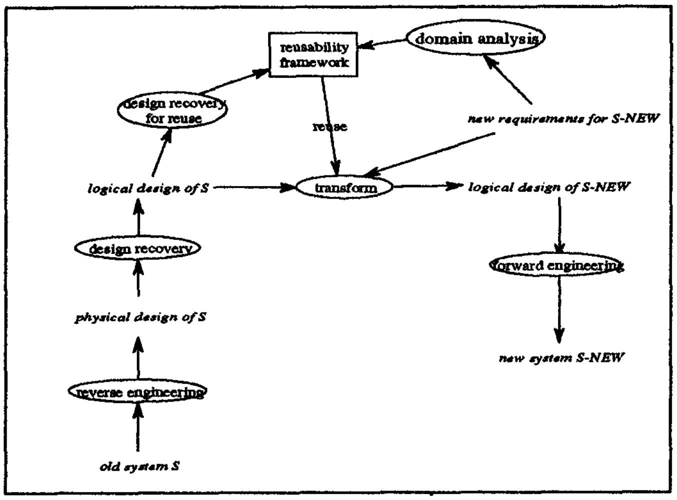

Fig. 2 Reengineering process model

The logical level provides the description of a program in terms of user-oriented, application domain concepts. The logical level may consist of Object-Oriented (OO) program descriptions[19], Entity-Relationship (ER) data models [5], Data Flow Diagrams (DFD), etc. The choice of representation for logical design depends on specific forward engineering techniques to be used as well as on programmer/user preference. Above logical program description level, there is a reusability framework that consists of a family of domain models and reusability management facilities. Each domain model describes designs and code that can be reused across systems in a given application domain such as payroll or customer service. (Application domains are also called business areas.) Domain models form OO program descriptions that are created in the course of independent domain analysis [1] and reverse engineering of systems that service a given application domain.

The technical scenario for software reengineering consists of three steps (fig. 3) that are performed at the application domain, system and system component levels, resp. The domain level step takes into account all the systems in a given application domain (AD). Objectives of this step are (1) to understand systems in AD, (2) to prepare an architectural framework for new systems (in particular, a common data model consistent with the target architecture), and (3) to do domain analysis in order to address problems of reusability.

The objective of the system level step is to produce complete logical design specifications for a selected system S in AD. Design specifications for both the original system S and target system (S-NEW) are produced. The design of S-NEW is based on a portion of a common data model relevant to S-NEW.

During the last step, a selected system is incrementally reengineered, component-by-component. Components may be individual programs or subsystems. Reusable features, accumulated within the domain model, are reused in component reengineering. As incremental reengineering of systems progresses, additional common features may be identified. They are extracted and linked into the domain model for future reuse.

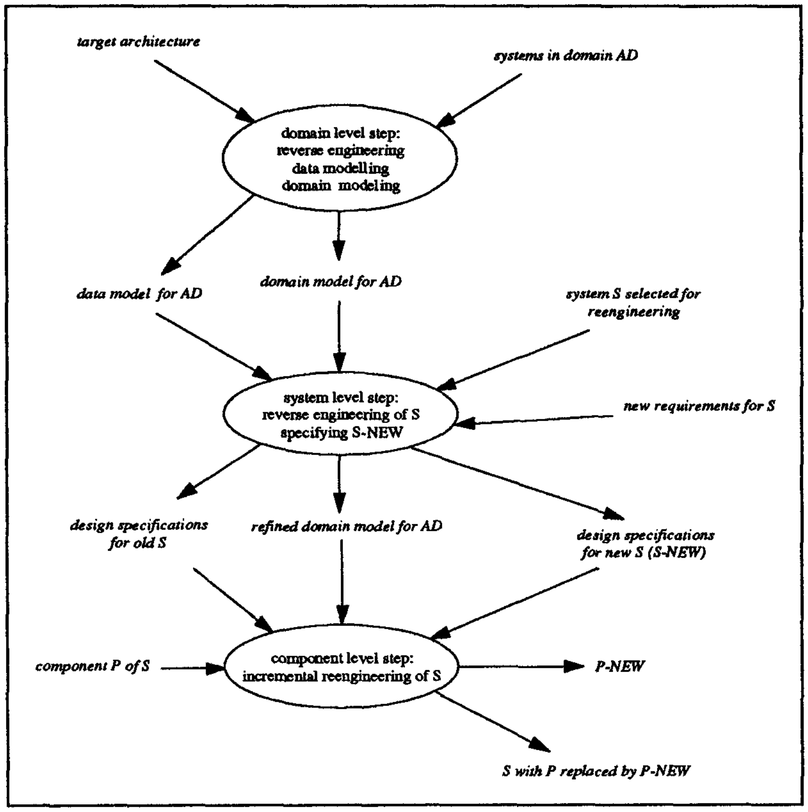
target architecture
systems in domain AD
domain level step: reverse engineering data modelling domain modeling
data model for AD
domain model for AD
system S selected for reengineering
system level step: reverse engineering of S specifying S-NEW
new requirements for S
design specifications for old S
refined domain model for AD
design specifications for new S (S-NEW)
component P of S
component level step: incremental reengineering of S
P-NEW
S with P replaced by P-NEW

Fig 3. Three levels of software reengineering

Domain models capture specifications and implementation of features that are common to all the software systems in a given application domain. But, apart from commonalties, we often observe that there are some variations in the way features are specified and/or implemented in different systems. In the reengineering context, those variations can be quite substantial. Suppose we reengineer system S to obtain system S-NEW. Having created domain models, we need to know how various features are specified/implemented in both systems S and S-NEW. This traceability of information from domain models to code is essential in software reengineering for reuse [12]. But some of the requirements for system S may no longer hold for S-NEW. Also, system S may be implemented in COBOL while S-NEW may be designed with CASE or built around an Object-Oriented architecture. Furthermore, software houses often maintain multiple implementations of a software package for different software/computer platforms. In such cases, platform-independent, logical model of reusable features (and of software packages) and explicit transformations from logical model into multiple implementations increase reuse potential and reduce maintenance effort. Therefore, multiple views of common features should be explicitly modeled to facilitate reengineering for reuse scenario. Fig. 4 depicts the architecture of such software models.

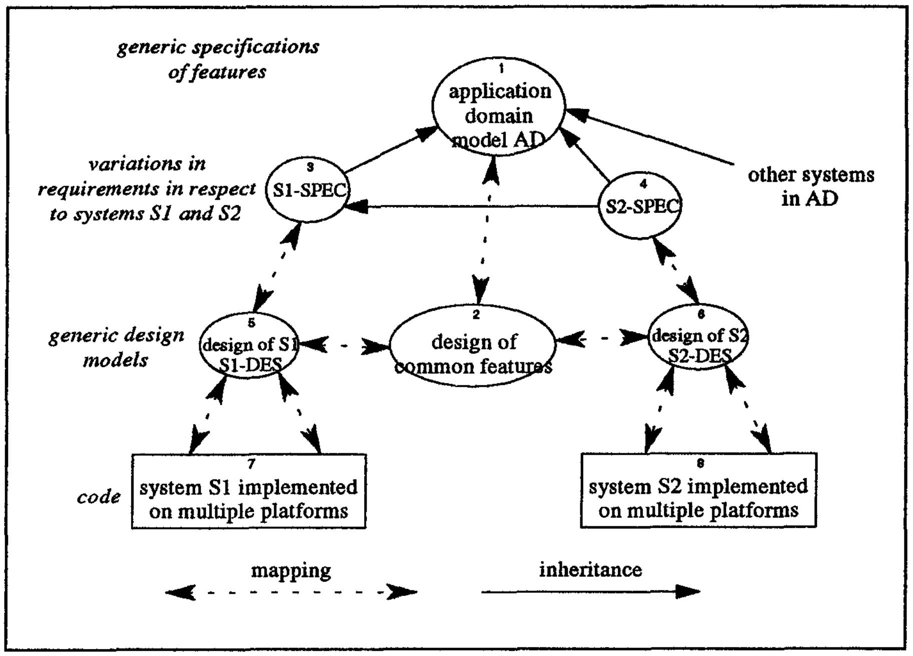
generic specifications of features
1 application domain model AD
variations in requirements in respect to systems S1 and S2
3 S1-SPEC
4 S2-SPEC
other systems in AD
generic design models
5 design of S1 S1-DES
2 design of common features
6 design of S2 S2-DES
code
7 system S1 implemented on multiple platforms
8 system S2 implemented on multiple platforms
mapping
inheritance

Fig. 4 An architecture of software models and mappings between models

Model 1 identifies common features in a given application domain AD and contains generic, user requirement level specifications of those features. Models 3 and 4 explicate variations in requirement specifications from the perspective of two systems in AD, S1 and S2. As we explain later in the paper, variations in requirement specifications across systems are modeled using inheritance. In the reengineering context, S1 can represent a system before reengineering and S2 - a reengineered version of that system. Models 5 and 6 contain system design specifications, expressed in terms of common features, in a platform-independent way. Finally, boxes 7 and 8 represent system implementations on multiple target platforms. In the remaining part of this paper, we concentrate on models 1, 3 and 4 in fig. 4, and describe a modeling technique of reusable features that is suitable in the reengineering context. In other papers, we described the role of domain analysis in reengineering [12], strategic reengineering lifecycle [13] and techniques involved in reengineering for reuse [14].

2. Modeling reusable features

We build domain models based on Object-Oriented approach (OO). Objects represent meaningful concepts from the application domain (e.g. an employee in a payroll system). In business programs, many of the interesting candidates for objects are naturally derived from a conceptual data model [12,19]. Objects comprise data models and procedures related to specific data groups. Modeling reusable features starts by reverse engineering of a conceptual data model based on analysis of data structures and database schema from existing programs. Reverse engineered data model is reconciled with new requirements for target systems and further refined in the process of data analysis. Next, object models are built by identifying procedures related to data model entities. Relationships between objects are derived from entity relationships. We have to stress that building an OO domain model to handle reusable features is helpful even if we do not intend to reengineer procedural programs into OO programs. If obtaining OO program architecture happens to be our objective, certainly an OO domain model will immensely help in such a transformation. But essentially, the main purpose of the object model is to organize program information for ease of understanding and reuse and to help in navigation through design/code during program reengineering and maintenance.

In our notation, program specifications are built around application domain *features*. Features refer to *objects* (e.g., a book in a library system), object *attributes* (e.g., an author), object *methods* (e.g., checking out a book), *relations* between objects (e.g., member Borrowed book), *events* (arrival of ordered books), global *procedures* and *business rules* (e.g., loan rules for various types of library users). Both object methods and global procedures form atomic *actions* that can be composed into *business processes*, i.e., chains of actions triggered by events. Business modeling methods similar to ours have been proposed by others [2,17]. In this paper, it is not our goal to demonstrate the modeling power of our notation. Instead, we concentrate on issues of how we actually represent and document features and variations in feature specifications. Features are described by *documentation templates*. A documentation template consists of *descriptors* grouped into *specification sections*. Each section has a title which is unique in a given template. Descriptors may be elements of formal specification, semi-formal or a free text. Descriptors may denote features and in such case they may refer to other documentation templates that describe those features in more detail. A descriptor consists of a *descriptor signature* (a name with optional list of arguments), followed by (an optional) descriptor body. Descriptor signatures must be unique in a specification section in which they appear. Documentation templates are organized into inheritance networks. The subject of inheritance are specification sections and descriptors.

As an example, we show how we document objects and methods. A documentation template for objects provides the following information:

- parent templates (in an inheritance network)
- a list of attributes
- for each attribute it may be specified:
    - attribute value domain and value constraints
    - whether attribute value can be changed or not
    - whether attribute is a key or not
    - whether attribute is computed or not
- a list of methods (methods are specified by separate method templates)
- object constraints (Boolean conditions)
    - invariants: characterize valid object states
    - initial: must be true for an object to be created
    - final: must be true for an object to be destroyed
- a list of rules (rules are specified by separate rule templates)

In a library system, we have library items such as books, journals, films, etc. Properties shared by all the library items might be defined in object template LIB-ITEM and templates for specific items might be derived from LIB-ITEM. Below we show documentation templates for features LIB-ITEM and BOOK:

domain object template LIB-ITEM {

informal description:

attributes:
CatalogNo
Title
int #copies = <1,Max>
Status = (Borrowed, Reserved, Available)

methods:
RegisterNew(LIB-ITEM)
CheckOut(LIB-ITEM,MEMB)
CheckIn(LIB-ITEM,MEMB)
BOOL IsReserved(LIB-ITEM)
BOOL IsBorrowed(LIB-ITEM)
BOOL IsOverdue(LIB-ITEM)

relations:
Borrowed(LIB-ITEM,MEMB)

object constraints:
IsAvail: Status = Available ↔ ~IsBorrowed(lib-item) & ~IsReserved(lib-item)

rules:
Overdue: If a LIB-ITEM is overdue more than one week, send a reminder to a borrower
if (IsOverdue(item)) **then** memb.Remind(item) **where** Borrowed(item,memb)
}

domain object template BOOK {

informal description:

derived from:
LIB-ITEM

attributes:
Author
ISBN
Status = (LIB-ITEM::Status, Reference)

methods:
CheckOut(BOOK,MEMB)
BOOL IsReference(BOOK)

rules:
Removal: If a book has not been used for 5 years, remove a book from library
}

Comments: Documentation template BOOK inherits all descriptor sections from LIB-ITEM. Method CheckOut is re-defined which means that specifications of CheckOut for books differs from CheckOut procedure defined in LIB-ITEM. In addition to descriptors inherited from LIB-ITEM, BOOK has a rule called 'Removal' and a method BOOL IsReference(BOOK). Attribute 'Status' is re-defined to reflect the fact that books can be placed on a reference shelf. 'IsAvail' is a signature of an object constraint that relates the value of attribute 'Status' to a condition expressed in terms of methods. (Symbol $\leftrightarrow$ means "if and only if".) The body of rule 'Overdue' contains both informal and formal specifications.

Method descriptors (in object template) may refer to method documentation templates that provide detail specifications of methods. In particular, method templates contain the following information (global procedures are documented in the same way as methods):

- method header: name, arguments and returned value
- objects involved in method; each object may be qualified as:
  - MODIFIED - if method modifies objects
  - INQUIRY - if method reads objects without changing them
  - CREATE - if method creates a new object
  - DELETE - if methods deletes objects
- pre-conditions: must be true before a method can be executed
- post-conditions: describes the effect of method execution

Here are documentation templates for methods CheckOut():

domain method template LIB-ITEM::CheckOut (LIB-ITEM item, MEMB b) {

informal description:

objects involved:
LIB-ITEM (MODIFIED), MEMB (MODIFIED)

pre-conditions:
Avail: ~IsReserved(item) & ~IsBorrowed(item) & MEMB::CanBorrow(b)

post-conditions:
NotAvail: IsBorrowed(item)
Borrowed: Borrowed(item, b)
}

Comments: The header indicates that this template refines a method descriptor CheckOut from template LIB-ITEM. The descriptors listed in LIB-ITEM can be used without qualification as long as this does not lead to ambiguous references. Descriptors from other templates must be qualified (e.g., MEMB::CanBorrow(b)).

Methods can inherit specifications one from another. In our example, method CheckIn is not re-defined in BOOK, therefore it applies to books. But specifications of documentation template for method CheckOut(BOOK,MEMB) slightly differs from method CheckOut(LIB-ITEM,MEMB), as books may remain on a reference shelf. To reflect this, we re-define pre-condition for method CheckOut(BOOK,MEMB):

domain method template BOOK::CheckOut (BOOK b, MEMB) {

informal description:

derived from:
method template LIB-ITEM::CheckOut(LIB-ITEM, MEMB)
pre-conditions:
Avail: LIB-ITEM::Avail & ~IsReference(b)
}

Comments: This example shows reuse of specifications across templates at low level of granularity: pre-condition 'Avail' defined in the parent template is used in definition of a stronger pre-condition in a derived template.

3. Modeling variations in feature specifications

The domain model is created not just for one system, but for all the systems in a given application domain. The domain model captures generic knowledge about an application domain, but there may be slight variations in requirements across systems. For example, a library may be located in several locations. It may happen that most sites allow users to reserve library items, but one site, say X, does not provide reservation service. Because of that BOOKs and method CheckOut will have different specifications in a system servicing site X from those that service other sites. Those variations must be traceable from the domain model down to design specifications and code in various systems under consideration. Differences between generic, domain model view and system-specific view of a given feature can be modeled by multiple inheritance.

We use the following conventions in modeling system-specific views:

1. a derived template must resolve any ambiguities resulting from multiple inheritance,
2. a template may hide certain elements inherited from parents,
3. an element hidden in template A cannot be accessed in templates derived from A,
4. a template can add new elements or re-define any inherited elements.

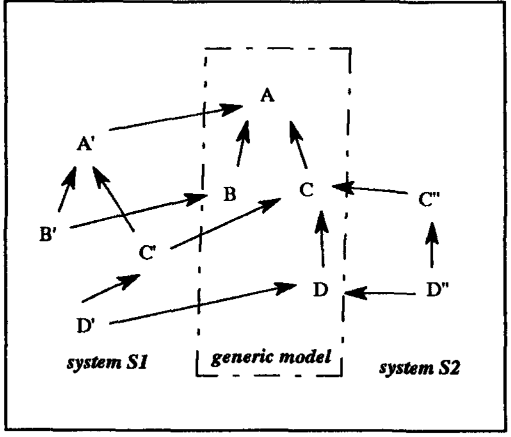

Fig. 5 Modeling system-specific views

In fig. 5, features A, B, C and D describe a generic model of application domain, say AD. S_1 and S_2 are two systems in AD. (In particular, S_1 might be a system before reengineering and S_2 - a reengineered version of that system.) System S_2 shares features A and B with its generic model. Feature C" is derived from C to show similarities and differences between system S_2 and generic model AD. As feature D" has some properties of D and some properties of C", it is derived from two parents. In system S_1, all the features are derived from the generic model.

To give a more intuitive illustration of a situation that involves modeling system-specific views, let's return to our library example. We model a view of library site X as follows:

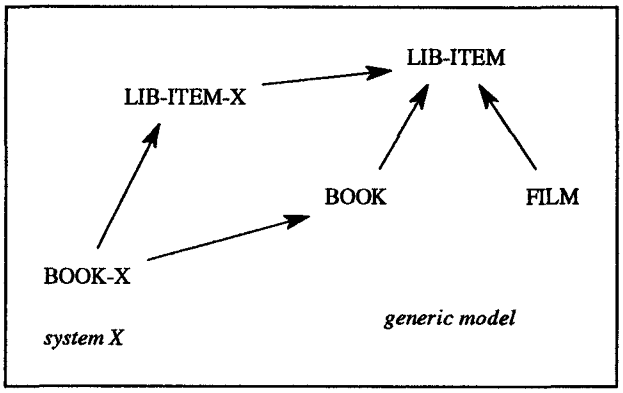

Fig. 6 System-specific views in a library system

domain object template LIB-ITEM-X {

informal description:

derived from:
  LIB-ITEM

attributes:
  Status = (Borrowed, Available)

...

methods:
  CheckOut(LIB-ITEM,MEMB)

hidden:
  BOOL IsReserved(LIB-ITEM)

}

domain object template BOOK-X {

informal description:

derived from:
  LIB-ITEM-X, BOOK

attributes:

```
    Status = (LIB-ITEM-X::Status, Reference)
methods:
    CheckOut(BOOK,MEMB)
hidden:
    BOOL IsReserved(LIB-ITEM)
}
```

Comments: Documentation template BOOK-X re-defines attribute 'Status' in terms of attribute inherited from LIB-ITEM-X, re-defines specifications of method CheckOut and hides method IsReserved. (It is necessary to hide method IsReserved in BOOK-X as it is inherited from two parents.)

In case of method CheckOut, we could derive a system-specific view and re-define the pre-condition (by deleting ~IsReserved(item) from the condition). But we also need to modify documentation template LIB-ITEM to reflect change in requirements from the point of view of X. Documentation templates for method CheckOut are derived in the following way (with pre-conditions modified to reflect no reservation service):

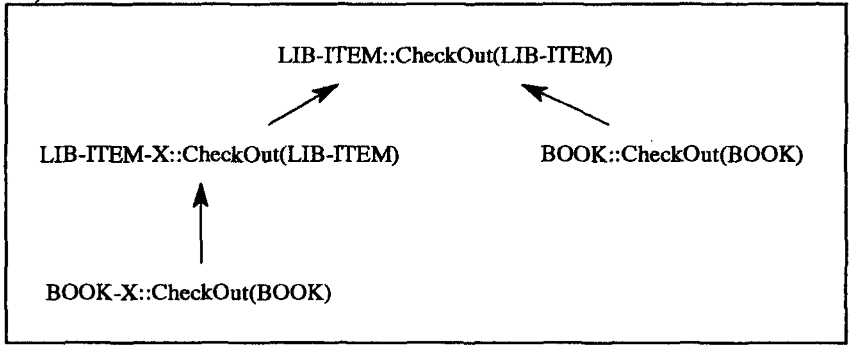

Fig. 7 Derivation of system-specific views of methods

We experiment with using a generator for language-based editors to support the above modeling notations. The generator can handle families of inter-related syntax trees. Each documentation template is represented by an attributed syntax tree and relationships between trees model inheritance. An incremental attribute propagation mechanism ensures semantic correctness of the domain model. We feel that more specialized environments should be built to support manipulation of OO domain models.

Design and implementation information is captured within the *design documentation templates*. Each domain feature may have an associated design template that explains how a given feature is implemented. A design template linked to a domain model feature provides generic implementation (stored in a library of reusable components), while a design template linked to a system-specific feature explains how a given feature is implemented in that system.

4. Related work

A number of authors identified a need for an explicit model to capture program design during reengineering and maintenance. Object-Oriented models for program understanding, based on application domain concepts, are described in [8,11,12].

During reengineering for reusability, programmer's task often is to isolate code that implements a given concept, to raise code to the logical level by removing implementation-dependent details and, eventually, to convert code into a reusable module. As it often happens in old programs, code that implements related concepts is not found in one program module, but is delocalized (i.e., spans a range of program modules). The process of finding and isolating that code can be greatly simplified with use of static program analysis tools [18]. Those tools can compress huge amount of code into a compact abstract view that is directly related to a certain aspect a programmer wants to study. Irrelevant details are filtered out of this view. Many useful program views are produced based on control and data flow relations. Particularly, program slicing views can automate the process of searching code that implements specific concepts. There are tools that compute program slices, extract them from programs and convert extracted code into a module, including necessary data declarations. Program slicing is an example of a technique that was first developed and experimented with in academic environments [7,22] and then successfully transferred into production use.

The program slicing technique, though very useful, provides only indirect means for recovering concepts behind programs. To address the problem more directly, we must explicitly model programming and application domain concepts and link them to relevant design abstractions and code. Automated program recognizers [9,10] attempt to define libraries of program plans that connect abstract concepts and their implementations. In the process of automated program recognition, a program is searched for instances of plans. As plans can be organized in a hierarchical way, the recognition process can progress from lower to higher abstract levels of program description. Today, most of the techniques for recovering reusable features are semi-automatic. If the results of research on automated program recognition scale up to real world programs, it may be possible that future tools will be able to control a bigger portion of the reengineering for reusability process.

Research on recovering object-oriented views from programs is also relevant to the reengineering for reusability. A method for identifying objects in C programs is described in [15]. Candidate objects are selected based on the analysis of type definitions; next, procedures which have arguments of a given type, or return a value of a given type, are identified as candidate methods. In [11], procedural programs are incrementally reengineered into an object-oriented architecture.

5. Conclusions

Many researchers and practitioners express opinion that software reuse has a potential to bring productivity breakthroughs and can fundamentally change the way we develop programs [3]. An important source of potentially reusable software assets are existing programs. Some of those programs, though they still contain much business value, will have to be reengineered, as they have become technically obsolete. To reengineer programs, we must understand them. Therefore, reengineering is an opportune moment to capture viable assets from existing programs and make them available for future reuse. Reengineering and reusability have potential to reinforce each other, but we need technical means to realize this potential. In work reported in this paper, we defined a mechanism for specifying common features in a given application domain and for modeling variations in feature specification/implementation across systems. Our specification method is suitable for the "reengineering for reuse" scenario.

We found it difficult to adopt one of the existing Object-Oriented systems to support the documentation resulting from domain analysis described in this paper. A system should be sensitive to the inheritance rules dealing with program specifications and should provide strong browsing capabilities. We are implementing a prototype documentation support environment using a structure editor generation system based on extended attribute grammars. Further work will also concentrate on adding more formality into specifications (based on notations proposed in [2]) and on modeling program dynamics.

References

1. Arango, G. "Domain Analysis - From Art Form to Engineering Discipline," *Proc. Fifth International Workshop on Software Specification and Design*, May 1989, Pittsburgh, pp. 152-159
2. Berztiss, A. "The Specification and Prototyping Language SF," Report No 78, SYSLAB, The Royal Institute of Technology, Sweden, 1990
3. Biggenrstaff, T. and Perlis, A. (Editors) *Software Reusability*, vol. I and II, ACM Press, 1989
4. Blum, B. "Documentation for Maintenance: A Hypertext Design," *Proc. of Conference on Software Maintenance*, 1988, 23-31
5. Chen, P. "The Entity-Relationship Model -- Toward a Unified View of Data," *ACM Transactions on Database Systems*, vol. 1, no. 1, 1976, pp. 9-36
6. Chikofsky, E. and Cross II, J. "Reverse Engineering and Design Recovery: A Taxonomy," *IEEE Software*, January 1990, pp. 13-18
7. Gallagher, K. "Using Program Slicing in Software Maintenance," TR CS-90-05, Ph.D. Thesis, University of Maryland, 1990
8. Hart, C. and Shiling, J. "An Environment for Documenting Software Features," *Proc. 4'th ACM SIGSOFT Symp. on Software Development Environments*, Irvine, USA, Dec. 1990, pp. 120-132
9. Hartman, J. "Technical Introduction to the First Workshop on Artificial Intelligence and Automated program Understanding," *Workshop Notes AAAI-92 AI & Automated Program Understanding*, July 1992, San Jose, pp. 8-31
10. Hartman, J. "Understanding Natural Programs Using Proper Decomposition," *13th International Conference on Software Engineering*, May 1991

11. Jackobson, I. and Lindstrom, F. "Re-engineering of old systems to an object-oriented architecture," *Proc. OOPSLA'91*, pp. 340-350
12. Jarzabek, S. "Domain Model-Driven Software Reengineering and Maintenance," *Journal of Systems and Software*, Jan. 1993, pp. 37-51
13. Jarzabek, S. "Strategic Reengineering of Software: Lifecycle Approach," 6th Int. Workshop on CASE, CASE'93, Singapore, July 1993, pp. 211-220
14. Jarzabek, S. "Software Reengineering for Reusability," *Proc. 17th Annual Int. Computer Software & Applications Conference, COMPSAC93*, Phoenix, November 1993, pp. 100-106
15. Liu, S. and Wilde, N. "Identifying Objects in a Conventional Procedural Language: An Example of Data Design Recovery," *Proc. Conference on Software Maintenance*, 1990, pp. 266-271
16. Martin, J. *Information Engineering*, Vol. 1, Prentice-Hall, 1986
17. McBrien, P. *et al* "A Rule Language to Apture and Model Business Policy Specifications," *Proc. 3rd Int. Conference on Advanced Information Systems Engineering CAiSE'91,* Trondheim, May 1991, Lecture Notes in Computer Science, no. 498, Springer-Verlag, pp. 307-318
18. Rock-Evans, R. and Hales, K. "Reverse Engineering: Markets, Methods and Tools," *Ovum Report* vol. 1, published by Ovum Ltd. England, 1990
19. Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F. and Lorensen, W. *Object-Oriented Modeling and Design*, Prentice-Hall, 1991
20. Strassmann, P. *The Business Value of Computers*, The Information Economics Press, 1990
21. Ulrich, W. "Re-development Engineering: Formulating an Information Blueprint for the 1990's," *CASE Outlook*, No. 2, 1990, pp. 15-21
22. Weiser M. "Program slicing," *IEEE TSE*, vol. 10, no. 4, July 1984, pp. 352-357