Conceptual modeling of families of software systems

Stan Jarzabek* and Tang Siong Tiing**
*Department of Information Systems and Computer Science
National University of Singapore
**School of IT, Nanyang Polytechnic

Abstract

Conceptual models must be sometimes built for a family of software systems rather than for a single system. Situations when this happens include evolutionary software development, software reengineering and reuse. In all those cases a conceptual model should characterize a given application domain and all the systems in that domain, e.g., payroll. For example, in software reuse, we look for reoccurring features (or concepts) such as objects, business rules, procedures, requirement/design patterns and code modules that can be reused across payroll systems. Capturing commonalties is a primary goal of a conceptual model. But apart from commonalties, we find that there are also some variations in feature specifications from one system to another. For example, some of the employee characteristics or rules for computing employee salary may differ from one payroll system to another. The premise of work described in this paper is that conceptual models should capture both commonalties and differences across a family of similar systems. In the reuse context, if variations in feature specification are not anticipated and explicitly modeled, customization of a component for reuse will not be easy: component customization will be done in an ad hoc way and programmers will have to go through the same customization process many times. Such reuse will be expensive and resulting systems will be difficult to maintain. These problems can be avoided if we explicitly model variations in feature specification. This paper shows how such multi-version models can be created. Conceptual models are composed of frames organized in Object-Oriented way. Our domain modeling method combines independent domain analysis and reverse engineering of low level design abstractions from existing programs.

1. Introduction

Conceptual modeling is an integral part of any systematic approach to software development and maintenance. The role of conceptual models is to capture our understanding of a software system in the form of formal or informal descriptions, and independently of the implementation: "...conceptual modeling should be regarded as a specification phase, where all the relevant knowledge about intentional aspects of the

application should be formally expressed and verified" and "the basic idea underlying conceptual modeling is to provide a high level formalism for specifying the information structure of an application, independently of the software environment used in implementation" [4].

Sometimes we build conceptual models for a family of software systems rather than for a single system. Situations when this happens include evolutionary software lifecycle, software reengineering and software reuse. We start by discussing them briefly.

In *evolutionary software lifecycle* [7], the difference between development and maintenance is blurred. A software product evolves over a sequence of versions, with each version addressing some new requirements. These versions form a family of systems that have a common core but differ in details from one another. To model an evolving software system, we must show what is common and what is different in respect to any two members of this family.

In *software reengineering* we also deal with families of similar systems. A source system before reengineering, say S1, and a target system after reengineering, say S2, show both commonalties and differences. Normally, basic functionality is not changed during reengineering, but some new requirements may be implemented in S2 and some the old requirements may get changed. Systems S1 and S2 may run on different platforms and be implemented using different implementation techniques. For effective reengineering, it is essential to have a single model from which we can trace commonalties and differences in user requirements and implementation details across the source and target systems.

To *software reuse*, we must understand a class of systems well enough to identify common features. Common features can then be made available to programmers either in the form of a library of reusable components (composition-based reuse [3]) or can be built into an application generation system (generation-based reuse). Apart from commonalties, we often observe that there are also variations in feature specification and implementation across systems [6]. For example, any payroll application will refer to employee, employee attributes (such as employee number, name, address, etc.) and procedures such as computation of employee's salary. However, payroll rules may differ from one department to another, e.g., in details of how to compute employee's salary or what information about employees should be stored. Some of the requirements that hold for one system may no longer hold for another system in the same application domain. Also, in case of packages developed by software houses, we may need to implement (and maintain) multiple versions

of the same system that can run on different software/hardware platforms. Then, a logical, platform-independent model of reusable features (and of software packages) and explicit transformations from logical model into multiple implementations increase reuse potential. A single logical model also substantially reduces maintenance cost, as future updates can be done to a logical model and then propagated to multiple versions of system implementation on different platforms. So in order to reuse, we need a conceptual model for a family of software systems in an application domain. This model should identify features common to all the systems in a given application domain, as well as variations in feature specifications and implementation across the systems.

An architecture of a conceptual model for a family of software systems is depicted in fig. 1. The architecture of a conceptual model consists of a generic part (models 1 and 2 in fig. 1) and a system-specific part (models 3 and 4 in fig. 1). The generic part of the architecture is built as follows. An application domain model captures features that are common across all the systems in a given domain. (Application domains are also called business areas [12].) Features common to all the application domains are further abstracted to form a company-wide model. For a well understood application domain, the domain model can be created prior to implementing systems. But for unexplored domains, we may need to implement one or more systems before we can build a domain model.

As an example, we show how the above conceptual model is built and used in software development with reuse. First, we analyze a given application domain, say payroll, as the whole. We take into account all the systems in a given application domain. Objectives of this phase are to understand systems in application domain, to identify reusable (i.e., common) features and to build a conceptual model of that domain. Suppose now that we implement a payroll system, say S1. First, we examine requirements for system S1 in the context of the domain model for payroll applications. Those parts of generic specifications from the domain model that directly apply to system S1 are selected. Then, we model differences between the domain model and S1 (model 3 in fig. 1).

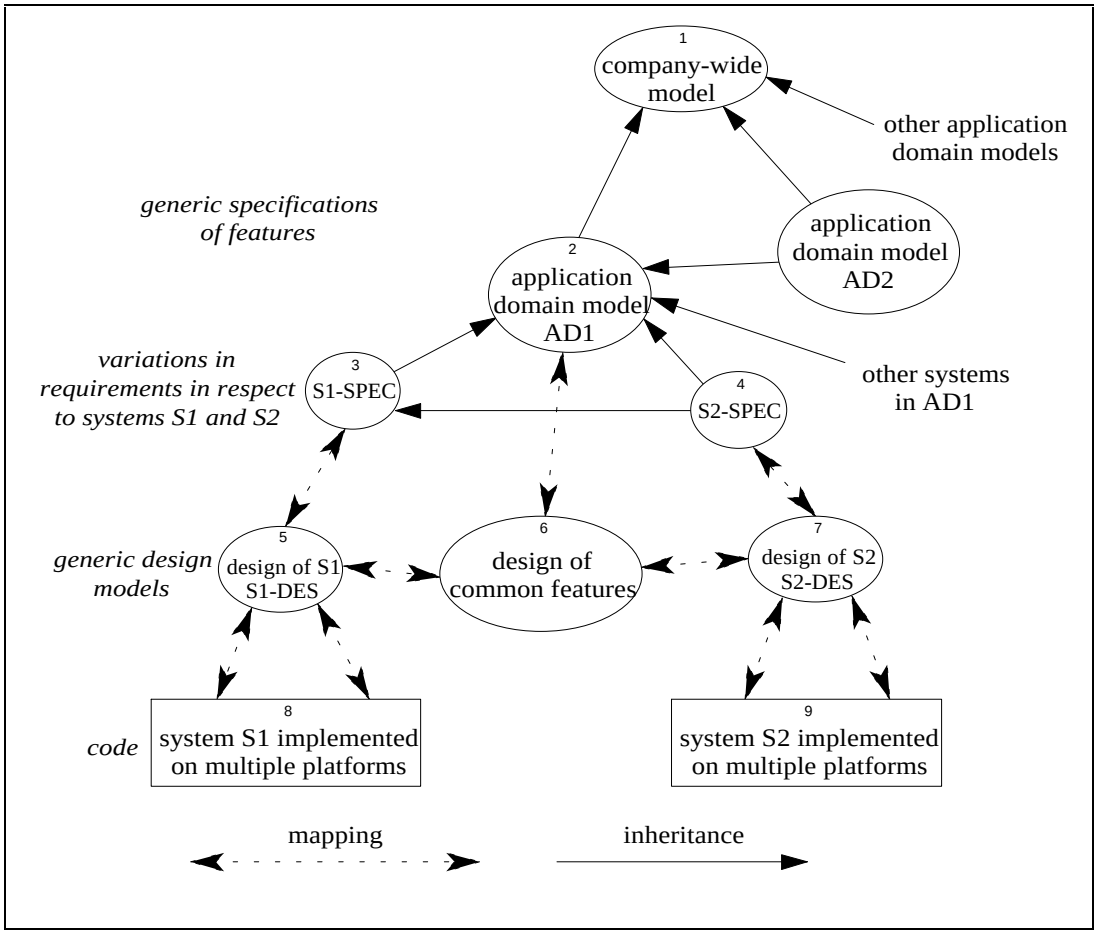

Fig. 1 Conceptual software model for reuse

If the same variations of requirements are found in two systems, say S1 and S2, a relevant part of S1-SPEC can be derived from S2-SPEC to make similarities between S1 and S2 explicit. Next, we produce logical design specifications for system S1 (model 5 in fig. 1). Design of system S1 is related to the generic feature design (model 6 in fig. 1) in a way that clarifies which features can be reused "as are", which features must be customized and which ones must be implemented from scratch. Parts of design models specialized for other systems (e.g., S2-DES) can be reused in building S1-DES. Finally, logical design of S1 is forward engineered into target platform(s). Forward engineering involves reuse of customized generic design of common features. Parts of code already specialized in building other systems can be reused in implementation of S1. In the above software development process, reuse may occur at the level of requirements (feature specification reuse), design (feature design reuse) and code (feature implementation reuse). As implementation of a system progresses, additional common features may be identified. They are included into software models for future reuse.

In the following sections, we first discuss the contribution of this paper in the context of other related work (section 2). Then, we describe our notation for conceptual modeling of families of similar systems (section 3) and show how we model variations in system requirements (section 4). In section 5, we provide an example showing how conceptual models facilitate reuse of design and code. Conclusion remarks end the paper.

2. Relation to other work

In this review, we concentrate on software reuse where conceptual modeling plays particularly important role [3,6,13,14]. Reuse that occurs early during software development life cycle is most beneficial and brings more leverage than reuse of code only. Conceptual models and generic software architectures facilitate reuse at levels of requirement analysis and design. Generally, reuse of code is not easy if we do not have an explicit model of a context in which code can be reused. Conceptual models and generic software architectures define such a context.

Reuse architecture in Draco[13] is based on software models that cover application domain level, design and implementation. Software models are hierarchically organized and are tightly integrated into a software development process with reuse. The need for multiple versions of software models arises naturally in evolutionary software development [7] and becomes particularly important in software reuse [6]. In reuse situation, we deal with variations in common feature specifications, design and implementation across a class of similar systems. To reflect these variations, we need to model multiple versions of software models at all abstraction levels, from conceptual domain models to code. The problem is of practical importance. If we do not explicate variations in common feature specification/implementation, reuse poses customization problems that are normally addressed in an ad hoc way. Because of that the main benefits of reuse, i.e. faster delivery of products and ease of future maintenance, are reduced. Gomaa [6] models variations in software requirements in generation-based reuse: target systems are automatically generated from system requirement and domain model specifications. The method described in this paper addresses modeling variations in software requirements for composition-based reuse. Our software model is built of Object-Oriented frames. Frames organize our knowledge of software systems for ease of understanding and reuse. Many modeling methods based on Object-Oriented concepts have been proposed in literature and our work builds on experiences described by others [1,2, 9,16].

The method described in this paper was developed and used in the context of modeling reusable information in software reengineering [9]. In software reengineering for reuse, modeling multiple versions of features becomes even more crucial than in software development with reuse. This is because a source system (i.e., one that we want to reengineer) and a target system after reengineering may differ a lot one from another. For example, reengineering may involve a drastic change of the implementation technique, such as from procedural to Object-Oriented design method or from 3rd generation language to program designed and generated with CASE. In strategic reengineering [10], requirements for target systems may also be different from those of source systems. Software models used in reengineering context must handle conceptual views of both source and target systems as well as mappings between them. The modeling method described in this paper is a continuation of work on modeling program information in reengineering for reuse presented in [9,10].

3. Modeling common features

The purpose of a conceptual model is to describe reoccurring features in a given application domain for understanding. Our conceptual modeling method and notation is designed for data-intensive business applications. We model software systems in terms of *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 can be related to atomic *actions* that constitute real-life *business processes.* These formal links between software and business models are essential in understanding the role of software systems in business environment. Fig. 2 depicts a business process that is composed of actions. Diamonds mark decision points at which alternative actions can be taken. Issues that affect the decision are modeled as conditions (cond 1 and cond 2 in fig. 2).

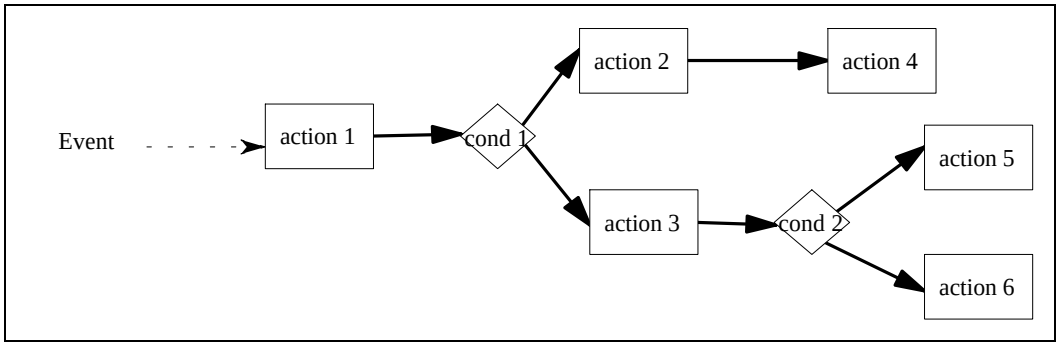

Fig. 2 A business process model

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 conceptual models. Let us first outline the process by which a conceptual model is created. Our approach to conceptual modeling combines both independent analysis of application domain and reverse engineering of lower level design information from existing systems. Motivation for such an approach is the following: we usually model application domains that we have already understood too some extent. Understanding a domain results from implementing a couple of systems in that domain. Those implemented systems contain invaluable information that can help us to observe common design and implementation patterns. Reverse engineering is to extract this information from code. But reverse engineering alone will not identify concepts. For this, we need an independent conceptual analysis step. Conceptual analysis tells us what is common while reverse engineering shows us examples of how common features are actually used in programs.

A central conceptual modeling feature is objects. Objects comprise data and actions related to specific data groups. In business programs, many of the interesting candidates for objects are naturally derived from a conceptual data model [9]. Modeling objects starts by reverse engineering an Entity-Relationship (ER) data model [5] from data structures and database schema in existing programs. Reverse engineered data model forms one of the inputs to conceptual analysis. During conceptual analysis, we first build an ER data model common to all the systems in a given application domain. ER data models reverse engineered from individual systems are mapped into a common ER data model for a given application domain. Next, specification objects are built around data model entities. We call them specification objects as they contain conceptual level specifications rather then code. Some of the relationships between objects are derived from entity relationships. Modeling further extends on object methods, object constraints and global procedures.

Within a conceptual model, all the features are described by *frames* that are organized into an Object-Oriented inheritance hierarchy. Subject of inheritance are specification elements, so the conceptual models can explicate commonalties and differences in feature specifications across application domains in a company and across systems within a given application domain.

A frame 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 frames 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. Frames 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 frame 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)

We shall explain our notation using, as a working example, a library information system. Here is a simplified description of a library system:

Library materials consist of books and films. There may be many copies of the same title in the library. Each title has a unique catalog number and each copy of a given title has a unique identification number. Books can be placed on the reference shelf. Reference items

cannot be loaned. Films cannot be placed on the reference shelf. Non-reference library materials can be reserved. Library users are academic members (students and staff) and external members. Loan terms are different for different classes of members.

The library information system stores data about library materials and members, records the status of each library item and helps library officers in checking out and in library materials, adding new copies into the library, and in other routine tasks.

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 frames for features LIB-ITEM and BOOK:

object LIB-ITEM

attributes {
informal-description: TEXT

CatalogNo : Cat-No

Title : TEXT

int #copies : (1:Max)

Status : (borrowed, reserved, available) }

methods {
RegisterNew(LIB-ITEM)

CheckOut(LIB-ITEM,MEMB)

CheckIn(LIB-ITEM,MEMB)

Reserve(LIB-ITEM,MEMBER)

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) }

end BOOK

*Comments***:** Frame for LIB-ITEM consists of five sections titled attributes, methods, relations, object-constraints and rules, respectively. Object constraint 'IsAvail' is formally

specified using logic notation. 'IsAvail' is a signature of an object constraint that relates the value of attribute 'Status' to a condition expressed in terms of methods. (Symbol ↔ means "if and only if".) The body of rule 'Overdue' contains both informal and formal specifications.

object BOOK

derived from LIB-ITEM

attributes {
informal-description : LIB-ITEM::informal-description | description specific to BOOKs

Author : Author-Name

ISBN-number : 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 }

end BOOK

Comments: Frame BOOK inherits all description sections from LIB-ITEM. Informal description of BOOKs consists of a generic description that applies to all library items (inherited from LIB-ITEM) and a description that is specific to BOOKs. BOOKs inherit all the methods from LIB-ITEMs. But method CheckOut is re-defined for BOOKs 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.

Method descriptors in object template may refer to method frames 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 is a frame for method CheckOut():

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

informal description: TEXT

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) }

end CheckOut

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 templates 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 frame 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):

method BOOK::CheckOut (BOOK b, MEMB)

informal description: TEXT

derived from: LIB-ITEM::CheckOut(LIB-ITEM, MEMB)

pre-conditions {

Avail: LIB-ITEM::Avail & ~IsReference(b) }

end CheckOut

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.

We have to stress that building an OO conceptual model is helpful even if we do not intend to use OO for building programs. Of course, if we choose to build systems in OO way, an OO conceptual model will help much in deriving OO architecture for systems that we develop. But essentially, the main purpose of the conceptual model is to present information for ease of understanding.

4. Modeling variations in system requirements

The conceptual model is created not just for one system, but for a family of systems in a given application domain. The conceptual model captures generic knowledge about an application domain, but there may be 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 conceptual model down to design specifications and code in various systems under consideration. We use multiple inheritance to model similarities and differences between generic, conceptual model view and system-specific views.

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.

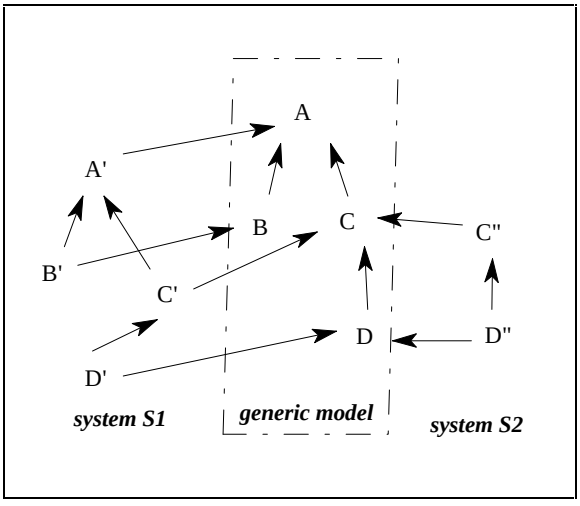

Fig. 3 Modeling system-specific views

In fig. 3, 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. 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. Fig. 4 depicts a view of library site X.

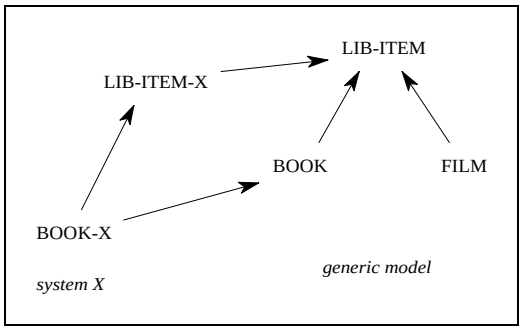

Fig. 4 System-specific views in a library system

```
object LIB-ITEM-X
    derived from: LIB-ITEM
attributes {
    informal description: TEXT
    Status : (Borrowed, Available) }
methods {
    CheckOut(LIB-ITEM,MEMB)}
hidden {
    BOOL IsReserved(LIB-ITEM) }
end LIB-ITEM-X

object BOOK-X
    derived from: LIB-ITEM-X, BOOK
attributes {
    informal description: TEXT
    Status : (LIB-ITEM-X::Status, Reference) }
```

```
methods {
   CheckOut(BOOK,MEMB)
hidden {
   BOOL IsReserved(LIB-ITEM) }
end BOOK-X
```

Comments: The frame 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 frame LIB-ITEM to reflect change in requirements from the point of view of X. Frames for method CheckOut are derived in the following way (with pre-conditions modified to reflect no reservation service):

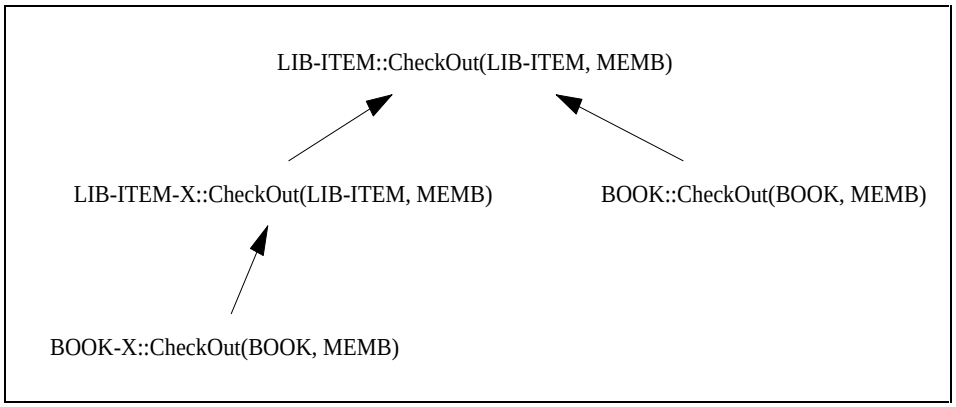

Fig. 5 Derivation of system-specific views of methods

We experimented with using a generator for language-based editors to support the above modeling notations [11]. The generator can handle families of inter-related syntax trees. Each frame 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. The strength of using a system like that for creating domain models is the ability to automatically maintain consistency constraints over the knowledge stored in models [15]. However, we feel that more specialized environments should be built to support manipulation of OO domain models.

5. Conceptual models in software reuse

As an example, we show how conceptual models of software families can facilitate software reuse. A generic design model for common features (model 6 in fig. 1) provides a bridge between conceptual model and implementations of common features. Similarly, a design model for system S1 (model 5 in fig. 1) provides a bridge between S1-SPEC and implementation of S1 on target platforms. Multiple design models for the same feature may be needed to reflect alternative design decisions. Feature design is documented by frames and mappings between feature design templates and corresponding feature specification templates are established.

Design models must be consistent with a software development method that we plan to use in software construction and must reflect the architecture of target systems. As the structure of application conceptual models is Object-Oriented (OO), it is most natural to continue with OO design and implementation. However, it is also possible to use other development methods and target architectures and still benefit from OO conceptual models.

In OO design, software architecture evolves around specification object models. In particular, data representation for objects is chosen and object methods are designed to facilitate implementation of required business processes. Method implementation is outlined in pseudocode. Target platform-dependent code segments are isolated and documented to simplify implementation on multiple platforms. Feature design is documented by frames that are linked to the corresponding feature specification frames.

As an example of an alternative development paradigm, we describe CASE-based design and construction that uses OO models to capture reusable features and system requirements. Modern CASE tools allow one to build software architecture around an ER data model. Software development process is also driven by a data model. Some of the standard development activities can be automated. For example, normalization of a data model, generating database schema into a target database, generating screens and reports can be done automatically. A programmer can attach validation rules and other processes to a data model and a data-driven CASE will generate proper code and embed it into a program. In such a development scenario, the first step is to design data models. In the Information Engineering methodology [12], three layers of a data model are created: company-wide data model, data models for individual application domain (or business areas) and data models for specific systems. These layers correspond to models 1, 2 and 3

in fig. 1. ER data models can be obtained from objects in conceptual models by mapping objects to entities, object attributes to entity attributes and object relationships to entity relationships. A company-wide data model can be derived from objects of a company-wide business model, an ER data model for a given application domain can be derived from an object model of that application domain and system-specific data models can be derived form models 2 and 3 (fig. 1). Actions (i.e., object methods and global procedures) are implemented using module structure available in a target language (e.g., procedures and functions in Pascal or programs and paragraphs in COBOL). All the processing units relevant to a given specification object can be grouped together to form object-like program modules. In that way, with certain limitations, we can benefit from a good modularity of OO design style without actually using an OO language for implementation.

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.

For example, design template BOOK of system X should show part of the ER data model that corresponds to BOOKs in X as well as logical processes that refer to that part of the ER data model. Design template BOOK should depict parts of programs that declare data structures for BOOKs, implement operations on BOOKs and store BOOK records in files. Mappings between object/entity attributes and corresponding record fields should also be provided.

6. Conclusions

We described a method for modeling families of similar systems. The method allows us to describe common features as well as variations in feature specifications across systems. We model application domains in terms of features such as events, objects, attributes, methods, object constraints, global procedures, business rules and business processes. Each feature is documented by a frame. Frames are organized into Object-Oriented inheritance hierarchies. Using inheritance, we model variations in feature specifications. Features common to all application domains are extracted into a company-wide business model.

We identified three situations in which the modeling method described in this paper is useful: evolutionary software lifecycle, software reengineering and software reuse. We used the method to model reusable information during software reengineering [9,10]. Our reengineering scenario assumes radical re-design of program structure rather than just restructuring or conversion. Such re-design is required when we change the software development technique, for example, from procedural to Object-Oriented development or from 3rd generation language to CASE-based development environment. We observed that it is possible to extract reusable features in the process of such reengineering. We built representation structures based on application domain concepts to capture reusable information. In this paper, we showed how similar structures can be used in general to describe families of software systems displaying variations in requirement specifications.

Future work will focus on automating our modeling method and on managing conceptual models. We found it difficult to adopt one of the existing Object-Oriented systems to support conceptual analysis described in this paper. A support system should be sensitive to the inheritance rules dealing with program specifications and should provide strong browsing capabilities.

Acknowledgments

Work described in this paper was supported by the NUS Research Grant No. RP920613.

References

[1] Auddino, A. “Formal Modeling of Objects and Subsystems in an Information Systems Framework,” in *Information Modelling and Knowledge Bases IV,* H. Jaakkola, et al., IOS Press, 1994

[2] Berztiss, A. "The Specification and Prototyping Language SF," Report No 78, SYSLAB, The Royal Institute of Technology, Sweden, 1990

[3] Biggerstaff, T. and Perlis, A. (Editors) *Software Reusability*, vol. I and II, ACM Press, 1989

[4] Catarci, T. and Lenzerini, M. “Conceptual database Modeling through Concept Modeling,” in *Information Modelling and Knowledge Bases III*, S. Ohsuga et al., Eds. IOS Press, 1992, pp. 59-74

[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] Gomaa, H. "An Object-Oriented Domain Analysis and Modeling Method for Software Reuse," *Proc. Hawaii Int. Conf. on System Sciences*, vol. II, 1992, pp. 46-56

[7] Gomaa, H. "Towards an Evolutionary Domain Life Cycle Model," *Proc. Workshop on Domain Modeling for Software Engineering, OOPSLA'89*, October 1989

[8] Jackobson, I. and Lindstrom, F. "Re-engineering of old systems to an object-oriented architecture," *Proc. OOPSLA'91*, pp. 340-350

[9] Jarzabek, S. "Domain Model-Driven Software Reengineering and Maintenance," *Journal of Systems and Software*, Jan. 1993, pp. 37-51

[10] Jarzabek, S. "Strategic Reengineering of Software: Lifecycle Approach," 6th Int. Workshop on CASE, CASE'93, Singapore, July 1993, pp. 211-220

[11] Jarzabek, S. "Specifying and Generating Multilanguage Software Development Environments," *Software Engineering Journal*, vol. 5, no. 2, March 1990, pp. 125-137

[12] Martin, J. *Information Engineering*, Vol. 1, Prentice-Hall, 1986

[13] Neighbors, J. "DRACO: A Method for Engineering Reusable Software Systems," In: *Software Reusability*, Biggenrstaff, T. and Perlis, A. (Editors) ,vol. I, ACM Press, 1989, pp. 295-320

[14] Prieto-Diaz, R. "Domain Analysis for Reusability," *Proc. COMPSAC'87, 1987, pp. 23-29*

[15] Tokuda, T et al. “Constraint Problems on Attributed Trees and Their Applications to Knowledge Maintenance,” in *Information Modelling and Knowledge Bases IV*, H. Jaakkola, et al., IOS Press, 1994

[16] Ueno, H et al. “Knowledge Modeling Based on Object Model - An Approach to Multi-Use Engineering Knowledge Base,” in *Information Modelling and Knowledge Bases III*, S. Ohsuga et al., Eds. IOS Press, 1992, pp. 75-96