Abstract: This article reviews a method for generating a product model from user data and introduces 12 design patterns for resolving conflicts that may occur in the integration and normalization process. The generating method and design patterns were evaluated during a project to integrate the product models collected from three different precast concrete companies in the United States, testing the new method of product model generation. The application results indicate that the 12 design patterns are effective and sufficient for resolving most known schema conflicts and for integrating and normalizing multiple product models into a single well-formed product model.

1 INTRODUCTION

The primary role of a product model is to support “communication, interpretation, or processing of product information” (ISO TC 184/SC 4, 1994a) between different systems. Depending on how the information exchange scenarios are defined, the scope and structure of a product (data) model vary. For example, the scope and structure of a product model for supporting product data exchange between different CAD systems will be different from those between CAD systems and resource management systems. Even between the same types of systems, different sets of product data can be exchanged depending on what information is required for a specified workflow. Thus, it is crucial to identify various data exchange scenarios in the early product modeling phase. As a means to model workflows and their product data use-cases, major standardization organizations today develop an application activity model (AAM) that relies on process modeling methods (IAI, 2005; ISO TC 184/SC 4, 1998; NIST, 1993). IDEF0 is the most commonly used in standard product modeling today (IDEF, 2002).

Ideally, one should explore and model all possible data exchange scenarios between different systems, based on the varying use-cases and the specific business practices of a range of organizations, then proceed to develop a product model based on the multiple process models (i.e., data exchange scenarios). However, numerous standard product models today including ISO 10303 STEP models, IFC extension models, and CIS/2, were developed based solely on a single unified process model (Crowley and Ward, 1999; IAI, 1994; ISO TC 184/SC 4,
The single unified process model may be helpful to understand the “overall” scope of a product model, but it is not detailed enough to identify what information is actually required in different data exchange processes. The unified process model also is not detailed enough to check whether a resultant product model can cover all targeted data exchange scenarios based on different company’s use-cases.

We are not aware of any product model now in use that is based on multiple data exchange scenarios. A likely reason is that there isn’t a method yet to specify detailed information items required for a process model and then to derive a product model from the specified information items. Conversely the product modeling community (e.g., ISO STEP, IAI, etc.) today spends much time and effort identifying subset product models, called conformance classes (Crowley, 2001) or information delivery models (IDMs) (Wix, 2005), from a final product model so as to explicitly define data sets required for specific information use-cases.

To overcome these drawbacks and to enable product modeling experts to derive a product model directly from multiple data use-cases, we developed a new product modeling method called GTPPM. The next section briefly reviews the GTPPM method and a tool developed to support the method. The basic concept, involving the logic rules to check the consistency of information flow within a process model, and an application result have been presented elsewhere (Eastman et al., 2002a,b; Lee et al., 2002; Sacks et al., 2002). This article focuses on the schema integration rules in GTPPM. It presents how multiple, independently developed partial product models can be integrated and normalized into a single model.

The integration and normalization rules we identify in this study are defined as design patterns (Alexander et al., 1977; Gamma et al., 1994). The integration rules were applied to a test-case in developing a single product model from three different information use-cases collected from three precast concrete companies. The evaluation results are presented in Section 5.

2 INTRODUCTION TO GTPPM

A new process-centric product modeling method, called the Georgia Tech Process to Product Modeling (GTPPM) method, was developed to facilitate domain experts to define detailed process models of their organizations, and then later to integrate the derived data into a single integrated product model. The goal was to provide product modelers with the logic and mechanism to reuse collected information and requirements for a product model in the logical data modeling phase and eventually to expedite the overall product modeling process. GTPPM is composed of three modeling phases:

- Requirements collection & modeling (RCM) phase: Domain experts specify various information use-cases (flow) and information items used in each use-case through this phase.
- Logical product data modeling (LPM) phase: (Semi-) automatically derive a product model from the information items collected from the requirements collection & modeling phase.
- Refinement phase: Elaborate and refine the resulting derived product model, including the integration of Integrated Resource libraries required for a fully compatible ISO10303 product model.

A review of commonly known software engineering methods (e.g., IDEF0, UML, (OMG, 2003), SSADT, Petri-Net, and DFDs), commercial computer-aided software engineering (CASE) tools (e.g., Visio, AllFusion, and Corporate Modeler), and software engineering methods and projects in the building industry (e.g., PISA (Bakkeren et al., 1996), OSMOS (Wilson et al., 2001), GPP (Wix and Katranuschkov, 2002), ISTforCE (Wix and Liebich, 2000), ATLAS (Tolman and Poyet, 1995), VEGA (Bakkeren et al., 1996), and ICCI (Katranuschkov et al., 2002)) revealed that although some methods (e.g., IDEF0, DFDs, PISA) had a capability of specifying information chunks (e.g., a file name) transferred from one point to another or to link a matured database to a process model (e.g., commercial CASE tools), none of them provided a mechanism to elicit a large set (typically over several hundreds) of distinct information items from domain experts and to structure them in a consistent format.

To specify the information items required by each activity in a process, GTPPM has specific information definition constructs and rules concerning the consistency of “information flow.” GTPPM is composed of 13 process modeling components (Figure 1). To describe the direction and the type of information flows, three types of information flow are defined: information flow, feedback flow, and material flow. Information flows pass information from one activity to another. Feedback flows are the iteration part of iteration cycles in information flows. Material flows represent the availability of physical materials for an activity, which may act as a hard-constraint or carry embedded information within the material. Other types of information flow include the continue shape that represents a syntactic bridge linking continued information flow. It can be used to break long information flow lines in a complex model into two pieces and to denote which flows are connected to each other (see Figure 2 for example). Two kinds of information repositories are defined: the static information source and the dynamic
information repository. While the dynamic information repository can store and dispense information during a project (i.e., within a process model), the static information source cannot be updated during a project and only can dispense information. Examples of dynamic information repositories are company databases, drawings, or reports. Examples of static information sources are company, regional, or national standards and libraries. Activities are categorized into four types according to the level of detail and the scope of interest: that is, the external high-level activity, the external detail activity, the internal high-level activity, and the internal detail activity. If an activity is within the scope of interest, it should be represented as an internal detailed activity. RCM modeling can be started by defining a sequence of high-level activities. Then, the high-level model can be elaborated further.
as sequences of detailed activities (similar to IDEF0). Note that the definition of level of detail and the scope are relative and are dependent on modelers’ intention and the particular universe of discourse.

Figure 2 is an example of a process of “Acquire Project” modeled collaboratively by precast concrete domain experts and product modelers.

GTPPM involves a collaborative work process between domain experts and product modeling experts (mediators). To facilitate their collaboration and to automate the procedures, the GTPPM software tool was developed and implemented as an MS Visio® add-on. The tool is designed to serve as a mediating tool where product modeling experts and domain experts jointly develop a data dictionary covering the information used in the domain of discourse, with which the domain experts then define their organization’s process models and information flows. The data dictionary must be flexible enough to support dialects (synonyms, local terms). Initially, the domain experts can specify information required by a process by specifying small chunks of information transferred from one activity to another.

The information required by each task (or activity) can be defined as local (user-defined) terms or expressions or as information constructs (ICs). An IC is an information item specified in a machine-processable format. It can be defined by concatenating several predefined terms following a close-to-intuitive formal syntax (Lee et al., 2006). An example is beam+section{type}. The asterisk (*) in an IC depicts the specialization (a.k.a., inheritance) relationship between the two concatenated terms: for example, piece+beam: a beam is a type of (precast concrete) piece. The plus symbol (+) depicts the decomposition relation or the association relation: for example, piece+feature: “a piece has a feature;” piece+fabrication: “A fabrication process has an association with the piece.” Items between a pair of curly brackets (‘{’ and ‘}’) are attributes of the last term in the concatenation. The terms are from a data dictionary as described above, which is jointly developed by the domain experts and product modelers. A long sentence such as “the length of a beam, which is a type of precast concrete piece” can be shortened as piece*beam{length}.

In GTPPM, a small chunk of ICs is called an “information set.” Some examples are shown in square brackets in Figure 2. Using the information sets as targets, specific information items that are used, generated, or processed by each activity can be specified. Also, the output information of each activity can be aggregated into information sets.

To derive a product model from information items collected from process models, all the user-defined terms must be mapped into equivalent ICs before the logical modeling phase.

In the logical product modeling phase, ICs are collected from different process models and integrated into a single model. Any conflict between ICs will be resolved through the integration and normalization process of the logical product modeling phase. The next section provides an overview of the integration and normalization method in GTPPM and Section 4 describes the integration and normalization rules, which are the main topic of this article, in detail. For more detail on GTPPM see Lee (2002).

3 SCHEMA INTEGRATION, NORMALIZATION, AND DESIGN PATTERNS

Sometimes different work processes use the same set of data. Sometimes two equivalent processes may use different sets of data. But usually different work processes require (and use) different sets of data. To make a product data model support various work processes, a product model should be an integration of different sets of data, which are required by different processes. We regard integration to be different from simple aggregation. Although aggregation is a simple consolidation of data, integration is a semantic union of different sets of data. In an integration process, semantic relations across data in different sets should be defined and mapped.

One issue is to identify any conflicts in an integrated model and to resolve them by normalizing the integrated product model into a well-formed model. Normalization is a process of decomposing and restructuring a data structure to achieve a higher “quality” or “goodness” of design (Elmasri and Navathe, 2004). Normalization of data was first proposed by Codd (1972) in the context of the relational model. Traditional normalization methods are based on data instances. It uses the known semantics of data in the form of dependencies that may be a cause for potential “update anomalies” requiring unnecessary duplicate work as well as causing potential inconsistencies in a database. The process successively decomposes the relations so that, after each decomposition, a higher normal form is realized; yet, the decomposition must be “nonadditive”—in that it does not produce any spurious data after joining the component relations. The relational normalization theory is well accepted and defines the first through fifth normal forms that consider null value, lossless join, and multi-valued dependencies (Elmasri and Navathe, 2004). It is difficult to apply the conventional normalization criteria and dependencies to some application domains: for example, the human genome databases (Kogelnik et al., 1998).

The definition of normalization in this article is not very different from most existing ones (i.e.,
decomposition and restructuring of a data structure to a normal form), but the scope and goals are not the same. Unlike traditional relational database normalization theories, the goal is not to eliminate redundancies or anomalies at an instance level, but rather remove them at the schema level. Because our focus of this study is on schema integration, instance-level normalization issues are out of its scope.

Schema integration can be regarded as a schema aggregation and normalization process. There are several ways of integrating (sub-)schemas. A brief and general introduction to product model integration methods is available in Section 6.3 of Schenk and Wilson (1994), for EXPRESS model integration. EXPRESS is the international standard product modeling language (ISO TC 184/SC 4, 1994b). Schenk and Wilson define integration as a process of combining topical information models (TIMs), which are domain-specific information models developed by several modeling teams, into a minimally redundant, nonambiguous and complete Integrated information model (IIM). The TIM and the IIM are conceptually similar to the ARM and the AIM of the ISO-STEP method (ISO TC 184/SC 4, 1994a) except for the fact that an IIM does not have predefined integrated generic resources (IRs). Schenk and Wilson categorize model integration into six forms:

1) cosmetic integration: modeling and documentation in a consistent style
2) editorial integration: elimination of synonyms and homonyms
3) continuity integration: elimination of redundancies and identification of gaps
4) structural integration: generalization of underlying concepts in TIMs and interfacing of an IIM with other IIMs
5) core-based integration: integration of TIMs into a high-level, abstract, and generic core information model
6) evolution-based integration: development of an IIM by integrating a TIM and another TIM until all the TIMs are integrated.

However, these descriptions provide only general guidelines and strategies for model integration and do not deal with the integration problems and solutions in detail.

Another effort worthy of discussion is EXPRESS-X. EXPRESS-X is the ISO STEP schema mapping language that provides a formal description method to map instance data between two different schemas using Rule Declaration and Type Map Declaration (ISO TC 184/SC 4, 1999). However, EXPRESS-X is a mapping mechanism between two schemas, not an integration method.

For some reasons, many people misconceive that XML can automatically integrate two or more schemas into a single schema. XML has Include, Import, and Redefine mechanisms to reuse or to integrate different schemas into a new one (Wyke and Watt, 2002). But they are not very different from the concept of the schema interfacing (or referencing) mechanism in EXPRESS.

Another approach to mapping and integrating (sub-) schemas is to use design patterns in object-oriented programming. Design Patterns originated from Christopher Alexander’s Pattern Language for buildings and towns (Alexander et al., 1977) and was later applied to software engineering. The design pattern for object-oriented programming has grown as a new field through years of efforts by design pattern groups and conferences (e.g., Ward Cunningham and Kent Beck (Coplien, 1999), Erich Gamma and his colleagues (Gamma et al., 1994), Pattern Languages of Programming conference, and Object-Oriented Programming conference, and Object Management Group). Each design pattern describes a particular object-oriented design problem; the core of the solution; and the constraints, consequences, and trade-offs of its use. In their book, Gamma et al. categorized three design patterns: creational, structural, and behavioral patterns (Gamma et al., 1994). The first two categories deal with instantiation and composition/decomposition of objects. The last category deals with encapsulation of algorithms. The proposed mappings in software patterns languages can be considered as best practices; they are revised in an evolutionary manner over time when better mappings are proposed. (Normal forms of relational databases can also be thought of as a representation of best practice; they were proposed and expanded over a 10-year period.)

The design pattern approach can be also applied to integration and normalization of a product model. However, in object-oriented data models, redundancy of data is less of a concern because of their efficiency of representing the specialization relationship and other relations using “pointers” compared to the relational data model, which relies on foreign key-primary key relationships. There have been several efforts to explore and develop normal forms for object-oriented data models that are different from relational normal forms (Beeri et al., 1978; Tari et al., 1997). They illustrate that object-oriented models can be decomposed and integrated relatively freely depending on the given normalization criteria. However, unlike relational normalization, the goals (or criteria) of normalization are not clearly set in object-oriented data modeling languages. For example, Tari et al. proposed user-interpretation-based normalization (Tari et al., 1997). And three functional dependencies (i.e., path dependency, local dependency, and global dependency) were provided to
support the method. Even though their method supports normalization (restructuring) of objects by user-defined constraints, the method is weak in terms of providing a standard or generic normalization method because any user-defined constraint can be a “norm.”

The main goals of the schema integration and normalization method in this study are (1) resolving conflicts between ICs with different data structures; and (2) eliminating redundant entities and attributes at a schema level.

4 DESIGN PATTERNS FOR SCHEMA INTEGRATION

This section describes 12 design patterns to resolve conflicts between different ICs—small chunks of information—and integrate them into a single model. Some patterns may be well known among the product modeling and object-oriented system community. However, they are included to make the collection of the patterns as complete as possible.

An IC is composed of entities and relationships between them. In this article, we use a simplified EXPRESS-G—a graphical form of EXPRESS—in representing ICs. Figure 3 is an example, which represents:

a. Entity SITE is an attribute of Entity PROJECT.
b. The relation between PROJECT and SITE is HAS.
c. Entity HOSPITAL is a subtype of PROJECT.
d. HOSPITAL may have a WARD. (A dotted line represents an optional relation.)

The relation between two entities is also called a “role” in EXPRESS and EXPRESS-G. In this article, the names of roles are omitted unless specified otherwise assuming that they are unique. Because EXPRESS is an object-oriented (or object-flavored) modeling language, an attribute can be an entity (type), a user-defined type, or a simple (attribute) type (e.g., numbers, strings, Boolean values, etc) unlike relational database languages. A user-defined type is literally a data type defined by a data modeler. To provide generic and less type-insensitive examples, we define attributes as an entity type in this article unless distinction is required. For full descriptions on EXPRESS-G, refer to (Schenk and Wilson, 1994) and (ISO TC 184/SC 4, 1994b).

The following sections describe 12 types of conflicts that may occur in the integration process and the design patterns used to resolve them. The general principle of model integration is to preserve semantics of ICs as much as possible in an integrated model so that an integrated model can support a wider range of data exchange.

Throughout the development of these patterns, it is assumed that the meaning of a term will be consistent across different process models that use it. In GTPPM, a glossary was provided with definitions of each term, and this rule was enforced to the degree possible.

In this article, design patterns are presented using a template composed of the following four parts.

- **Design Pattern Name**: a unique name that represents the contents of the design pattern.
- **Problem**: description and examples of a conflict between data definitions.
- **Solution**: a conflict resolution method.
- **Notes**: additional comments on related cases. This part is optional.

The design patterns introduced in this section were validated and refined through the evaluation process of three test cases described in Section 4 using schema validation tools in EDM.®

Some mathematical symbols used in the definitions for the design patterns are

- $P \rightarrow Q$: If P is true, then Q is also true
- $P \lor Q$: P or Q
- $P \land Q$: P and Q
- $A \cup B$: the union of A and B
- $A \cap B$: the intersection of A and B
- $A \supseteq B$: A subsumes B. Or B is a subtype of A.
A ≡ B: A and B are the same set.

x ∈ A: x is an element of A.

x ⊢ A: x may be an element of A.

∀x : for all x.

∃x : there exists x.

where P and Q denote a proposition, A and B denote a set, and x denotes an element of a set.

4.1 Design pattern 1: A conflict between attribute definitions

Problem: Different ICs denote that an entity has different attributes (Figure 4).

In different use cases, the same entity may be defined with different sets of attributes. In such cases, no attribute should be abandoned. For example, assume that Entity PROJECT is one context that describes the activities on site, and the other is defined in the context of the relationship with its client, as shown in examples P and Q below:

P: PROJECT ≡ {NAME, JOB_CODE, SITE}
Q: PROJECT ≡ {JOB_CODE, MANAGER, SCHEDULE, CLIENT}

Let’s assume Entity A in Figure 4 is PROJECT in this example. Entity C is SITE. Entity D is SCHEDULE. To integrate them as a single product model that can support both cases, the two data sets must be unionized first. As noted earlier, as we are assuming that the tokens used in these data sets are from a single data dictionary and follow the “nym” principle (i.e., no homonym, no synonym), tokens with the same spelling can be regarded as identical. Thus, the union of P and Q is

P ∪ Q : PROJECT ≡ {NAME, JOB_CODE, SITE, MANAGER, SCHEDULE, CLIENT}

This rule is generalized as follows.

Solution: Each relation has a specific meaning in structuring a data model. Thus, the attributes of an entity should be the union of attributes defined in separate ICs. This rule also implies that if an entity is defined as an attribute of another entity in one case and no relationship was defined between these two entities in another case, the relation should be kept in the integrated model. The rule can be formerly written as follows:

If one IC defines that Entity R has x as an attribute and another construct defines that Entity R has y as an attribute, both x and y should be regarded as attributes of R:

x ∈ R ∧ y ∈ R → R ≡ {x, y}

If we apply this rule to the conflicting construct examples illustrated in Figure 4, the resolution will be as shown in Figure 5.

Notes: If there is a case where different attributes of an entity are associated with one entity type as shown in Figure 6a, there is no conflict in the IC. An example is where A is “schedule”; Role_1 is “start_date”; Role_2 is “end_date”; and C is “date” in Figure 6a. On the other hand, there should not be a case illustrated in Figure 6b where one role name is associated with two attributes if tokens are defined following the “nym” principle. However, if such cases occur, one of the Role names must be changed. An example is where A is “product”; both Role_1 and Role_2 are “identifier”; C is “product_serial_number”; and D is “product_type_code.” In this case, either Role_1 or Role_2 should be renamed.

4.2 Design pattern 2: An attribute shared by two entities in a chained relationship

Problem: In EXPRESS, an entity can be an attribute of another entity. Thus, there can be a case where an attribute of an entity can be defined as an attribute of the same entity. For example, in Figure 7, Entity C is an attribute of Entity A in the left-hand side case. But in another case, Entity C is also defined as an attribute of Entity B, which is an attribute of A (the right-hand side case in Figure 7).

Solution: This case falls into a case where two different entities share one attribute, which is not a conflicting situation. For example, two entities PRODUCTION and
SHIPPING may share the same attribute DUE_DATE. This is possible and there is no conflict in this case. In addition, even if SHIPPING is also defined as an attribute of PRODUCTION (i.e., production information is associated with shipping information) as shown in Figure 8, the relations are still valid.

4.3 Design pattern 3: Entities in a bidirectional association relationship

Problem: In one case, Entity A is defined as an attribute of Entity B. In another case, Entity B is defined as an attribute of Entity A (Figure 9).

Solution: This is a very common case and both association relations should be kept in an integrated model as a bidirectional relationship. An example is the inverse relation. The definitions of “normal” and “inverse” are relative. For example, the relation from PRODUCT to PART is “is_composed_of” (Figure 10). And the inverse relation is “composes.” From PART to PRODUCTION, the normal relation is “composes” and the inverse relation is “is_composed_of.” One of them could be represented specifically as the inverse relation of the other in EXPRESS and EXPRESS-G. However, the
bidirectional relation between two entities is not always the inverse relation. For example, two entities BUILDING and PERSON may have a bidirectional relation, which is not the inverse relation: BUILDING “is owned by” PERSON. PERSON “uses” BUILDING. Whether the bidirectional relation is the inverse relation or not, the relations in both directions should be kept.

4.4 Design pattern 4: Optional versus mandatory relations

Problem: The relation between two entities is defined as the normal (mandatory) relation in one case and as the optional relation in another case (Figure 11).

Solution: A relation acts as a semantic constraint between two entities. To enable a product model to support more information use-cases, the constraint should be as inclusive as possible. As the normal relation is included in the optional relation, the optional relation should be kept in an integrated model. That is, \( x \in A \rightarrow x \vdash A \) (i.e., \( x \in A \) or \( x \notin A \)). However, the following rule does not hold: \( x \vdash A \rightarrow x \in A \).

Notes: If the lower boundary of the cardinality between two entities is set to zero, such cases should also be regarded as the optional relation.

4.5 Design pattern 5: A conflict between subtype definitions

Problem: The specialization relation between two entities is represented in one case, but not in another case. (Figures 12a and b). Also different cases (ICs) denote that an entity has different subtypes (Figures 12a and c).

Solution: Specialization in Figure 12a denotes that all the attributes and relations in Entity A are also in Entity B, although in Figure 12b the figure shows them as disjoint. Similar to the solution of Design Pattern 1, all the subtype relations should be kept in the integrated IC (Figure 13) with the default DISJOINT subtype constraint so that the collected entities and relations can be maintained.

4.6 Design pattern 6: A duplicate subtype relation

Problem: In the above example illustrated in Figure 13, if there is an additional case where Entity C is defined also as a subtype of Entity B (Figure 14), it yields a case where an entity (e.g., Entity C) is defined as a subtype of two entities (e.g., Entities A and B), which are already in the supertype-subtype relationship.

Solution: Because all the attributes of a supertype will be inherited to a subtype through a hierarchical structure, it is redundant to define an inheritance relationship between a supertype and a subtype when the subtype is already associated with its supertype through a hierarchical structure. Mathematically, a supertype always carries a subset of a subtype’s attributes and \( (C \supseteq B) \land (B \supseteq A) \rightarrow C \supseteq A \). In the example above, the
Figure 15. A resolution for a duplicate subtype relation.

Figure 16. A logically impossible bidirectional supertype-subtype relation.

The supertype-subtype relationship between Entities A and C is already implied through the relationship between Entities A, B, and C, and, therefore, redundant. The redundant relationship must be eliminated as shown in Figure 15.

Notes: In automated product model construction, it is very unlikely that there is a bidirectional supertype-subtype relationship between two entities as illustrated in Figure 16. But if such relations are constructed as a result of IC integration, theoretically the two entities must be the same entities. This is because, if A ⊆ B and B ⊆ A, then A ≡ B. If they are the same entities, one entity should be renamed as the same name of the other entity and the two entities should be merged. Or if they are two different entities, one of the subtype relations should be removed. No logic to choose one name over the other has been identified yet.

4.7 Design pattern 7: A conflict between attributes of a supertype and attributes of its subtypes

Problem: The same set of attributes belongs to a supertype and to all its subtypes. For example, in Figure 17a, Entity A has two Entities L and K as attributes. Another case illustrated in Figure 17b shows that the Entities L and K belong respectively to Entities B and C, which are subtypes of Entity A.

Solution: [Case 1] Because all the attributes of a supertype will be inherited to its subtypes, it is redundant to define the attributes of a supertype again as attributes of its subtypes. In Figure 17, if Entity A already has Entity K as an attribute, it is redundant to define Entity K as an attribute of Entity D, which is a subtype of Entity A. The redundant attributes of subtypes should be deleted in an integrated model as shown in Figure 18. That is, if B ⊇ A and x ∈ A, then x ∈ B. However, (B ⊇ A) ∧ (x ∈ B) → x ⊥ A; i.e., x may or may not be an attribute of Supertype A and the proposition x ∈ A may not hold. Thus, in such conflicting cases, the relation between x and Subtype B should be removed and the relation between x and Supertype A should be kept.

[Case 2] Similarly, if both relations (the relation between an attribute and a supertype and the relation between the same attribute and a subtype) are defined as optional, the relation between the attribute and the supertype should be kept. That is, if Entities C and D are subtypes of Entity A and x is an optional attribute of A, the proposition that x will always be an optional attribute of both Entities C and D will hold:

(C ⊇ A) ∧ (D ⊇ A) ∧ (x ⊥ A) → x ⊥ C ∧ x ⊥ D
Twelve design patterns

Figure 18. Deletion of inherited attributes.

Figure 19. An entity defined as an optional attribute of both a supertype and a subtype in an inheritance tree.

However, if Entities C and D are subtypes of Entity A and x is an optional attribute of C, x may not always be an optional attribute of Entity A.

If \((C \supseteq A) \land (D \supseteq A) \land (x \vdash C)\), then \(x \vdash A\) may not hold.

Figure 19 illustrates an example of this additional rule. The example includes the following propositions:

a) EMPLOYEE may receive a bonus.
b) MANAGER may receive a bonus.

In this example, Proposition a) subsumes Proposition b). However, Proposition b) does not guarantee the validity of Proposition a).

[Case 3] Alternatively, if Entities K and L are defined as optional attributes of Entity A, and are also defined as a mandatory attribute of respectively Entities B and C as shown in Figure 20 below, then the Entities K and L should be regarded as the attributes that distinguish the two subtypes B and C from each other. In such cases, the optional relations between K, L, and A should be removed and the mandatory relations between K and B and between L and C should be maintained to represent the distinction between B and C. This rule can be generalized as follows: if \(B \supseteq A\) and \(x \in B\), then \(x \vdash A\): that is, the optional relation between \(x\) and A is kept. On the other hand, if the relation is defined in a reversed way \(B \supseteq A\) and \(x \vdash A\), then the proposition \(x \in B\) does not hold: that is, the mandatory relation between \(x\) and B will not be maintained. Therefore, in such cases, the optional relation between \(x\) and Supertype A should be removed and the mandatory relation between \(x\) and Subtype B should be kept. An example is illustrated in Figure 20. The example includes the following three propositions:

a) EMPLOYEE can be paid either by a flat rate or hourly.
b) HOURLY WAGER, a type of EMPLOYEE is paid hourly.

Figure 20. Deletion of optional attributes of a supertype in an inheritance tree.
c) FLAT-RATE WAGER, another type of EMPLOYEE is paid by a flat rate.

In this case, if only Propositions b) and c) are kept, then Proposition a) will be always satisfied. However, if only Proposition a) is kept, Propositions b) and c) will not be always satisfied. Thus, Propositions b) and c) should be kept.

[Case 4] Let us assume that there is a case where Entities K and L in Figure 20 are defined conversely as mandatory attributes of Supertype A and as an optional attribute of respectively Subtypes B and C at the same time. In such cases, also the mandatory relation, but on the supertype side this time, should be kept. The logic is

\[(B \supseteq A) \land (x \in A) \rightarrow x \in B\]

\[x \in B \rightarrow x \dashv B\] from Design Pattern 4

\[\therefore (B \supseteq A) \land (x \in A) \rightarrow x \dashv B\]

However, if \(B \supseteq A\) and \(x \dashv B\), the proposition \(x \in A\) does not hold.

Notes: The difference between Cases 3 and 4 of Design Pattern 7 and Design Pattern 4 is that Design Pattern 4 is a conflict between an optional relation and a mandatory relation in one entity whereas the examples given in Cases 3 and 4 are conflicts caused by the inheritance mechanism between an optional relation in a supertype entity and a mandatory relation in a subtype entity.

4.8 Design pattern 8: Generalization

Problem: Different subtypes of a supertype have common attribute(s). For example, in Figure 21, Entities B and C, two subtypes of Entity A, have the same attribute D.

Solution: By definition, a supertype is a set of least common attributes of its subtypes:

\[\text{Supertype } T \equiv \{x: \text{attribute}; S: \text{subtype of } T \mid \exists x \forall S(x \in S)\}\]

Thus, the common attribute(s) should be deleted from the subtypes and moved up to the first common supertype as exemplified in Figure 22.

Notes: The difference between this pattern and the above Design Pattern 7 is that Design pattern 7 deals with a conflict between attributes of subtypes and attributes of their supertype whereas this design pattern defines a pattern for creating new attributes of a supertype by extracting least common attributes of its subtypes.

4.9 Design pattern 9: A conflict between the subtype relation and the association relation

Problem: The relationship between two entities is defined as the association relation in one case and as the supertype-subtype relation in another as illustrated in Figure 23a and b.

Solution: Two entities can be both in the attribute relation and in the specialization relation. For example, Entity ASSEMBLY can be an attribute and a subtype of Entity PIECE at the same time (Figure 24). (NB: In implementation, it is better to define PIECE as an attribute of ASSEMBLY considering their many to one relationship.) PIECE composes ASSEMBLY; ASSEMBLY is a type of PIECE. Thus, in this case, both relations should be kept.
4.10 Design pattern 10: The attribute data type conflict

*Problem:* In EXPRESS, data types such as BINARY, STRING, NUMBER, REAL, INTEGER, LOGICAL, BOOLEAN are called *simple types*. In this case, an attribute is defined as an entity type in one IC and as a simple type in another (Figure 25).

*Solution:* An entity can have its own attributes and, thus, carries more information than a simple type or a (user-) defined type. Thus, in this case, the entity-type attribute should be kept and the simple-type attribute should be eliminated (Figure 26). A (user-) defined type cannot have an attribute, but its name and structure is still user-definable (e.g., it also can be an enumeration of other types), the order of selection among EXPRESS data types should be:

Entity > (User-) defined types
> Simple (attribute) types

4.11 Design pattern 11: A conflict between simple attribute types

*Problem:* Different ICs define the data type of an attribute as different simple types (Figure 24).

*Solution:* The order of selection of simple types should be dependent on the expressiveness of data types in EXPRESS. For example, REAL can be expressed by NUMBER, but REAL cannot express NUMBER. INTEGER can be expressed by REAL, but REAL cannot express INTEGER. LOGICAL can be expressed as 1, 0, −1 in INTEGER, but LOGICAL cannot express INTEGER.

Fig. 24. A resolution for the subtype and attribute conflict.

Fig. 25. A conflict between an entity type and a simple type.

Fig. 26. A resolution for the attribute data type conflict.
REAL can be expressed by STRING, but REAL cannot express STRING without converting it into a human-illegible format. Thus, we can say that STRING is more expressive than REAL. Thus, the STRING type attribute should be kept in the given example (Figure 28). BINARY is an exceptional case. Any data can be converted to BINARY, but the converted data loses its semantic meaning. Thus, if there is a case where a simple data type conflicts with the BINARY type, it is better to define the data as a non-BINARY type to keep the semantic. The order of expressiveness of simple data types is

String > Number > Real > Integer

> Logical > Boolean(> Binary)

4.12 Design pattern 12: A conflict between existential constraints

Problem: The existence constraints between two entities are defined differently in two different ICs as exemplified in Figure 29.

Solution: EXPRESS has four existential constraints: three dynamic aggregation constructs (namely BAG, LIST, and SET) and one static aggregation construct (ARRAY). In EXPRESS-G, they are represented respectively as B, L, S, and A using the initial letter. The lower and upper bounds of the potential number of elements in each aggregation is defined within the square brackets: for example, S[0:10], L[1:?], B[n:m] where the question mark symbol denotes an indefinite upper bound. As ARRAY is static, the size is fixed: for example, A[10]. A LIST is ordered. A SET does not allow duplicate elements. A BAG is unordered and allows duplicate elements.

A conflict between existential constraints can be examined from two points. First, there may be a case where the aggregation type is the same, but only the lower and upper bounds are different. In such cases, the lowest and highest bounds should be taken so that the definition can cover a wider range of cases. For example, if there are two different bounds [3:?] and [1:10], the bound of an integrated IC should be [1:?]. However, if the conflicting ranges are discrete, for example, [0:2] and [5:?], such cases should be flagged and the reason for the discrepancy should be examined.

If aggregation types are not the same, a more inclusive aggregate type should be chosen. BAG is the most inclusive among the four aggregation types. In theory, any conflict between different aggregation types can be resolved by defining the aggregation type as BAG. However, except for BAG, it is not trivial to judge which one is more inclusive than others. ARRAY can be interpreted as a type of BAG with the same lower and upper bounds: A[m] = B[m:m]. In practice, SET and LIST are the most commonly used aggregation types. In the case of LIST and SET, because the definitions of LIST and SET are not mutually exclusive and when one constraint is converted to another, it significantly diverts the meaning of an IC, the inclusiveness between LIST and SET cannot be judged. For example, the identifiers of an object must be unique. Thus, they can be defined as a SET, not as a LIST. On the other hand, the vertices of a polygon must be defined in a certain order. Thus, they must be defined
Fig. 30. A resolution for the existence constraint conflict.

as a LIST, not as a SET. If there is a conflict between SET and LIST, it is likely that either one of the definitions is invalid although the design pattern introduced here is based on an assumption that all the collected ICs are valid. Such conflicts between SET and LIST should be flagged and examined. If we apply these rules to the example illustrated in Figure 29, the resolution will be as shown in Figure 30.

5 DEPENDENCIES BETWEEN THE 12 DESIGN PATTERNS

Thus far, we described 12 design patterns for resolving conflicts that may occur in the integration process of data sets collected from various use-cases. There are dependencies between these design patterns and the dependency must be taken into account in implementation of these patterns. First, several design patterns propose a union of seemingly conflicting cases as a solution. The patterns that fall into such a category are as follows:

- Design Pattern 1: A conflict between attribute definitions
- Design Pattern 2: An attribute shared by two entities in a chained relationship
- Design Pattern 3: Entities in a bidirectional association relationship
- Design Pattern 5: A conflict between subtype definitions
- Design Pattern 9: A conflict between the subtype relation and the association relation
- Design Pattern 4: Optional versus mandatory relations
- Design Pattern 6: A duplicate subtype relation
- Design Pattern 7: A conflict between attributes of a supertype and attributes of its subtypes
- Design Pattern 8: Generalization
- Design Pattern 10: The attribute data type conflict
- Design Pattern 11: A conflict between simple attribute types
- Design Pattern 12: A conflict between existential constraints

Design Patterns 4, 10, 11, and 12 are about the conflicts between entities in the association relation. Design Pattern 6 is about the conflicts between entities in the specialization relation. Design Patterns 7 and 8 are about the conflicts between entities in both the association relation and the specialization relation. Thus, Design Patterns 4, 10, 11, and 12 and Design Pattern 6 are mutually exclusive, but Design Patterns 7 and 8 are dependent on all other design patterns. Hence, Design Patterns 7 and 8 should be implemented last. As Design Patterns 7 and 8 are independent of each other, the implementation order is insignificant between these two. Among Design Patterns 4, 10, 11, and 12, Design Pattern 4 is dependent on Design Pattern 12 because an option role is a special case of the existential constraints where the lower bound of an existential constraint is set to 0. Others are independent of each other. Table 1 summarizes the implementation order of 12 design patterns.

These 12 design patterns have been deployed in a test case to build a single product model from three different data sets. The test case is presented in the next section.

6 APPLICATION OF DESIGN PATTERNS

These design patterns were deployed in integrating three product models derived from two different management processes and one designing/drafting process of precast concrete products. The three processes were respectively collected from three different precast concrete companies in the United States: Company A (Denver, PA), Company B (Springboro, OH), and Company C (Pittsfield, MA) in years 2002 and 2004.

In the process models, we also defined a list of specific information required by each activity in the processes. As a result, we were able to derive 135 distinctive ICs...
Table 1
The implementation order of 12 design patterns

<table>
<thead>
<tr>
<th>Order</th>
<th>The association relation</th>
<th>The specialization relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><strong>Design Pattern 1</strong>: A conflict between attribute definitions</td>
<td><strong>Design Pattern 5</strong>: A conflict between subtype definitions</td>
</tr>
<tr>
<td></td>
<td><strong>Design Pattern 2</strong>: An attribute shared by two entities in a chained relationship</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Design Pattern 3</strong>: Entities in a bidirectional association relationship</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Design Pattern 9</strong>: A conflict between the subtype relation and the association relation</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td><strong>Design Pattern 10</strong>: The attribute data type conflict</td>
<td><strong>Design Pattern 6</strong>: A duplicate subtype relation</td>
</tr>
<tr>
<td></td>
<td><strong>Design Pattern 11</strong>: A conflict between simple attribute types</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Design Pattern 12</strong>: A conflict between existential constraints</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td><strong>Design Pattern 4</strong>: Optional vs. mandatory relations</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td><strong>Design Pattern 7</strong>: A conflict between attributes of a supertype and attributes of its subtypes</td>
<td></td>
</tr>
</tbody>
</table>

from Company A, 231 distinctive ICs from Company B, and 85 distinctive ICs from Company C. Major entities collected from Company A are listed below as examples:

ASSEMBLY
BIDDING
BOM
BUILDING_CODE
CONSTRAINTS
DIMENSIONS
ENGINEERING
EQUIPMENT
ERECTION
GEOMETRY
MOLD
OC_CHECK
SHIPPING
SURFACE_TREATMENT
TRUCK_LOADS
DESIGN REQUIREMENTS
DOCUMENTATION
DRAWING
ERECTION_DRAWING
ESTIMATION
HARDWARE
HARDWARE_LIST
LABOR
MATERIAL
PIECE
PIECE_DRAWING
PIECE_LIST
PRODUCTION_AND_HANDLING
POUR
PRESTRESSING
PROJECT
REINFORCEMENT
SCHEDULE
SITE BATCH (mix recipe)
CONCRETE (mix recipe)

The major differences between two business management processes from Companies A and B and a designing/drafting process from Company C, in terms of information flow, is that information flow in the designing/drafting process is accumulative: that is, a model of a precast concrete structure behaves as a data repository. As soon as a designer adds one shape or text to a precast concrete model or to a drawing, they represent certain information. Also such design information does not affect only the activities immediately following it, but also many other activities that appear later in the process.

Another difference between the Company A and Company B models and the Company C model is that the process, types of pieces were defined by generic information such as product name or piece-mark, whereas, in the Company C designing/drafting model, types of pieces were defined specifically as spandrel, pc_column, floor_piece, and so on. To design a piece, designers need to know which type of piece (at an object level) is connected to which other type of piece. For the same reason, even though we only focused in the process modeling within Company C on the processes of designing/drafting two piece types (double tees and spandrels), the definitions of adjacent pieces and connections were also captured in the derived product model.
ICs collected from the above three models were automatically parsed and integrated as a single product model through the 12 design patterns defined earlier. The 12 design patterns were implemented in Visual Basic. The integrated model was composed of 129 entities. For reference, CIS/2 LPM 6 (Crowley, 2003) has 731 entities and PCC-IFC Version 0.9 (Karstila et al., 2002) has 413 entities.

Figure 31 illustrates an EXPRESS-G model of the automatically integrated PIECE and CONNECTION definitions from Company A, Company B, and Company C without any modification. In this particular model, we intentionally did not define data types and the existential constraints in the early modeling phase assuming that they will be refined later. In such cases, the system automatically assigns STRING as the default data type.

The syntax of the automatically generated integrated product model was validated using two schema validation tools (those embedded in the commercial tool EXPRESS Data Management (EDM®) Supervisor Version 4.5 and in the shareware Expresso Version 3.1.4). These schema validation tools can evaluate not only the basic level syntax (e.g., whether there is a parenthesis at the beginning or the end of a certain sentence or whether reserved terms are used inappropriately), but also most of the anomalous and conflicting cases described in the 12 design patterns in the previous section (e.g., conflicts between entity definitions, conflicts between entities and attributes, etc.). The integrated model passed both validation processes successfully.

To ensure the validity of the integrated model, the model was again tested by implementing it as a physical database on MS SQL Server 2000® and also on EDM® “as-is.” (For the MS SQL implementation, GT-EXPRESS2SQL® was used to convert EXPRESS code to SQL code.) If there is any anomalous and conflicting data definition in a schema, these systems do not create a physical database and return an error message. The application results and our experiments with other smaller models indicate that the 12 design patterns can resolve conflicts between different schemas and can integrate multiple schemas into a single well-formed product model. That is, a single product data model can be generated by integrating several partial product models without compromising multiple process models (information use-cases).
12 design patterns that semantically interpret information depicted in a process model and resolve conflicts that may occur in the integration and normalization process of product models in EXPRESS have been defined. The 12 patterns have been implemented in a tool and tested. Each design pattern defines a rule to resolve a specific type of conflict between different ICs (a piece or a chunk of information). The principle of these conflict resolutions is to select a product model construct that is semantically “inclusive,” that is, a construct that can carry information of all the conflicting constructs without loss, so that an integrated product model can cover all its subset models.

The 12 design patterns are based on the following two preconditions:

a) The terms used in ICs must be uniquely defined, each with a single definition.
b) All the ICs in the collected data sets must be semantically valid.

In this study, we restricted the product model derivation by requiring that all the ICs must be reducible to a single set of semantically consistent tokens from a single data dictionary. Throughout several test cases, we have tried several other approaches such as adding synonyms to a data dictionary or categorizing tokens by general knowledge representation concepts (e.g., specialization, aggregation association, etc.). We found that those approaches are potentially effective at a certain level. However, these add to the learning and consistency requirements of the domain experts and the verification of these semantics within a process modeling tool remain to be developed. Requirement (b), that the semantic constructs used to create a product model be semantically correct, emphasizes the subtle differences in the interpretation of some design patterns (e.g., Design Patterns 6 and 7) depending on the definitions of collected ICs and the need that the domain experts are able to make these distinctions in the ICs. These capabilities have been only partially satisfied. As an effort to guarantee the validity of collected ICs, we have earlier proposed an approach to check the validity of information by evaluating the consistency of information flow (Lee et al., 2002). The fundamental logic is to check the availability of information from one activity to another. However, today there is no method that can completely detect and eliminate semantic errors.

The 12 design patterns were evaluated through a project to integrate three product models collected from three different precast concrete companies in the United States. The application results indicate that the 12 design patterns are effective and sufficient for resolving schema conflicts in this test case and for integrating and normalizing multiple product models into a single well-formed product model. However, in the same way as the number of normal forms for relational database gradually increased, in the future additional design patterns for product model schema integration and normalization may be identified and added to or substituted for these 12 patterns. This study mainly focused on product model integration, but the application of these design patterns is not limited to the product model integration. They can be applied also to other types of object-model data model integration and normalization efforts.

The integration process outlined logically incorporates the use-cases generated by different companies. We have not yet addressed, however, the extraction of company-specific use-cases for application back into a company’s information exchanges. This work is still to be undertaken. Also, in generalizing some structures during the normalization (merging) process, we have loosened their semantics as defined in the generating organization’s use-cases. It may be possible to capture these and re-apply them when the use-cases are later extracted. Such capabilities open the door to developing stronger semantic models than is possible currently, by linking them to specified use cases. Although our original intent was to reduce the length of time required to develop a product model, and to put the development of them on a stronger methodological base, it may be that schema integration, as presented here, also can provide opportunities to improve the semantic clarity of product data models based on ISO-STEP technology.

ACKNOWLEDGMENTS

We thank the referees for their helpful, insightful, and detailed comments and Dr. Shamkant Navathe at the College of Computing, Georgia Institute of Technology for his comments on the early draft of these design patterns.

REFERENCES


Twelve design patterns


Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley.


NIST. (1993), FIPS Publication 183: Integration Definition of Function Modeling (IDEFO), National Institute of Standards and Technology.


