EVOLVE: Adaptive Specification Techniques for Object-oriented Software Evolution *

Ling Liu

Department of Computing Science
University of Alberta, 615 GSB
Edmonton, T6G 2H1 Alberta, Canada
Email: lingliu@cs.ualberta.ca

Abstract

The increased complexity of object-oriented models necessitates the enhancement of adaptiveness and robustness of an object-oriented design towards changing requirements. The understanding of what properties are critical for construction of an adaptive schema design becomes increasingly important in software evolution. In this paper we present two groups of techniques for enhancing the adaptiveness and the robustness of an object-oriented design in anticipation of future requirement changes. The first group of techniques consists of a selection of adaptive schema style rules for achieving validity, minimality, extensibility and normality of a schema design. We encourage to use this set of rules as a means for validating quality of a schema, and for transforming an object-oriented schema into a better style, in terms of adaptiveness and robustness of a schema design, rather than as a user-oriented method solely for designing the schema. The second group of techniques includes the use of propagation patterns and propagation pattern refinement. Propagation patterns are employed as an interesting specification technique for modeling the behavioral requirements. They encourage the reuse of operational specifications against the *structural* modification of an object-oriented schema. Propagation pattern refinement is suited for the specification of reusable operational modules. It promotes the reusability of propagation patterns towards the *operational* requirement changes. The main innovations are in raising the level of abstraction for behavioral schema design, and for making possible the derivation of operational semantics from structural specifications. We argue that, by using these adaptive specification techniques, the workload required for reorganization and reprogramming of the existing investment (object base and programs), after parts of the system have been changed, can largely be avoided or minimized.

*This work was initiated when the author was with the department of Computer Science at University of Frankfurt.

1 Introduction

Object-oriented software systems, like all software systems, are not constructed in "one-shot", but rather developed in an evolutionary way by prototyping models of the software systems and by restructuring and reprogramming during their whole life cycles [BD91]. As the number of software applications increases and the costs associated with maintaining and adapting them to changing requirements escalate, the interest in software evolution grows. A recent study [**?**] found that 60 to 85 percent of the total cost of software is due to maintenance. Moreover, only 20 percent of maintenance consists of fixing bugs, so-called corrective maintenance. Another 20 percent goes to adaptive maintenance, which results from changes in the software environment. The majority of maintenance efforts (about 60 percent) is invested in so-called perfective maintenance, involving continued development and evolution of a software system after it has become operational. Therefore, an evolution phase is becoming an important part of the life cycle of a successful software product [**?**]. The following are two of the main issues in software evolution:

- How does one maintain a system's structural and behavioral consistency after parts of the system have been changed?
- Can one improve the design and implementation of a software system to gain better adaptiveness and robustness in anticipation of future requirement changes?

In fact, these two issues also represent two closely related and complementary research directons in object-oriented software evolution. One direction emphasizes on effective management of change impact on both an object-oriented schema, the object base, and the corresponding software programs after parts of the software system have been changed. The ultimate goal is to adapt and to propagate the changes to the affected parts of the system to maintain the structural and behavioral consistency of the system. The other direction is targeted at developing techniques for improving the adaptiveness and robustness of the program design or the structural and behavioral sepcifications of the system. As [BH93] have shown, program adaptation can be exceedingly hard for typed languages (such as C++) even for simple schema changes. It is therefore important and benefical to devote some research effort on enhancing the adaptiveness and robustness of a schema design or a program specification so as to avoid or to minimize the workload for change propagation and program adaptation in the presence of schema updates, rather than trying to fix things after changes occured.

The EVOLVE project mainly contributes to the second direction of research and development in the context of object-oriented database specifications. We assume that both the schema and the database programs in an object-oriented software system may be updated (or evolved) after the database is populated with object instances and application programs have been implemented and tested. Therefore, the impact of schema modifications implies not only the propagation of restructuring operations into the database instances, but also the reprogramming of existing application programs (e.g., relevant methods and queries). The main objective of the EVOLVE project is two-fold.

- First, we develop a selection of style rules for construction and evaluation of an adaptive schema, aiming at improving the adaptiveness and robustness of an object-oriented schema design in anticipation of future requirement changes.
- Second, we argue for avoiding or minimizing the amount of effort required for manually reprograming of existing method and query specifications due to schema evolution or requirement

modifications. We demonstrate how the reuse mechanisms can be exploited for enhancing the adaptiveness of database programs in the presence of schema evolution.

Why an adaptive schema design becomes increasingly desirable?
An object-oriented schema is adaptive and robust if it correctly uses the object-oriented modeling concepts, and can easily be adapted to changing requirements by encouraging as much information localization and data abstraction as possible, so that the overall impact of a schema modification or a database update can be minimized or reduced. For instance, a schema modification should be localized at one place whenever possible. No update anomalies should be incurred due to a modification to the database. Due to the increased complexity in both the schema design and the development of application programs working with the schema, any change to the existing schema design may have effect on both the database instances and the application programs. The cost for change propagation and program adaptation can be exceedingly high, when the system is large in size and complex in organizational or behavioral dimension. So far, most research in object-oriented design has mainly concentrated on how to use object-oriented concepts in logical design and requirement modelling. There is, however, surprisingly very little on evaluation of what constitutes a robust and adaptive schema in anticipation of future requirement changes. Many desired properties of a schema design (such as validity, extensibility, minimality, normality) (cf.[**?**]) are only vaguely understood in the context of object-oriented models.

With these objectives in mind, we develop a selection of design and evaluation rules for building an adaptive schema in an object-oriented system. We encourage to use the set of style rules proposed as a means for validating quality of a schema and for transforming an object-oriented schema into a better style in terms of adaptiveness and robustness, rather than as a method solely for designing the schema. One of our primary goals is to automate quality design and evaluation for building an adaptive object-oriented schema, such that we may use this set of style rules not only to verify the many desired properties of an object-oriented schema design, but, if the schema is found to be not or less qualified, these style rules can also be used as baselines to transform the schema into a better style and yet semantic-preserving alternative. The schema transformation algorithms should, on the one hand, eliminate the undesired dependencies and minimizes the impact of schema modifications, and, on the other hand, be able to preserve all the necessary dependencies and thus the database integrity constraints required by applications during schema enhancement process and schema change management.

Can reprogramming be avioded in the presence of schema evolution?
Reprogramming of object methods and database queries usually follows evolutionary changes of the logical object structure (i.e., the database schema). For example, in most existing method definition or query specification languages, each name used in methods or queries must be associated with a precise path expression in order to traverse the nested structure of the objects. Whenever a schema modification involves more than one existing class, the path expressions relevant to those classes are changed in the modified schema. The methods and queries which use those "old" path expressions must be updated accordingly to enable them to be valid in the modified schema. Operations for reprogramming of methods and queries can be quite expensive, especially when the relevant application programs are large and complex. Moreover, these operations conflict with the reuse of software components. Interesting to note is that, quite often, the reason for manually reprogramming of methods and queries after schema modifications is simply to keep the path expressions required in method definitions or query specifications consistent with the modified schema. In such cases, manually reprogramming of existing methods or queries (due to schema modification) can certainly be avoided by using structural derivation of operation propagation semantics, because the precise knowledge of path expressions is

actually derivable from the logical object structure of the corresponding schema. There are many other cases where manually reprogramming of existing methods or queries (due to schema modification) can also be avoided in a similar way, for example, when a schema modification changes the properties of objects (e.g., a new property is added to an existing class). Unfortunately, very few object-oriented systems (and none of the existing object-oriented DBMS products we know of) include support for structuring and deriving operational semantics from structural specifications. We believe that adding support for automatically or semi-automatically deriving the semantics of operation propagation over the hierarchical structure of complex objects opens new possibilities for the reuse of operational specifications (such as methods or query programs) in object-oriented systems. Of course, when a schema modification has substantially updated the logical object structure of a schema (in particular, when a schema modification changes the minimal knowledge required for specifying a method or a query), the reprogramming cannot be avoided completely [**?**].

In the EVOLVE project, we propose a seamless approach to the incremental design and reuse of object-oriented methods and query specifications and show how the reuse mechanisms are exploited for improving the adaptiveness of software programs against schema modification in an object-oriented system. We argue that, by using this approach, operational specifications become more robust and adaptive towards schema modifications. The effort to manually reprogram methods and queries necessitated by schema modifications can be avoided or minimized. The salient features of this approach are the use of *propagation patterns* and a mechanism for *propagation pattern refinement*. Propagation patterns can be seen as an interesting specification formalism for modeling operational requirements in any object-oriented system. They encourage the reuse of operational specifications against the *structural* modification of an object-oriented schema. Using propagation patterns provides method designers and query writers with an opportunity to specify operations without detailed navigational information. Propagation pattern refinement is suited for the specification of reusable operational modules. It promotes the reusability of propagation patterns towards the *operational* requirement changes. We provide a number of examples to illustrate the concepts of propagation patterns and propagation pattern refinement, and to show why these concepts are important reuse mechanisms and how they are employed to avoid or to minimize the effort required by manually reprogramming of methods and queries due to schema modifications.

The rest of the paper proceeds as follows. We first briefly present the EVOLVE reference object model in Section 2. Then we introduce the set of style rules the EVOLVE tool provides for construction and evaluation of an adaptive schema in an object-oriented software system in Section 3. Due to the space limitation, in this paper we only report in detail the formal semantics of the style rules for correct use of inheritance as a representative example, and discuss the rest of rules informally. In Section 4, we present the concept of propagation patterns and their polymorphic character as well as the importance of behavioral refinement of propagation patterns in improving adaptiveness of object-oriented behavioral specifications. We provide an overview of the EVOLVE methodology and implementation framework in Section 5. The paper end by a comparison of the EVOLVE development with the related work and a summary in Section 6.

2 The Reference Object Model

We use the kernel of the Demeter data model [LX93] as our reference object model because this allows us to show how the reuse mechanisms such as propagation patterns and propagation pattern refinement can directly be made available using an existing tool: the Demeter System/C++^{TM} ([LR88], [Lie94]).

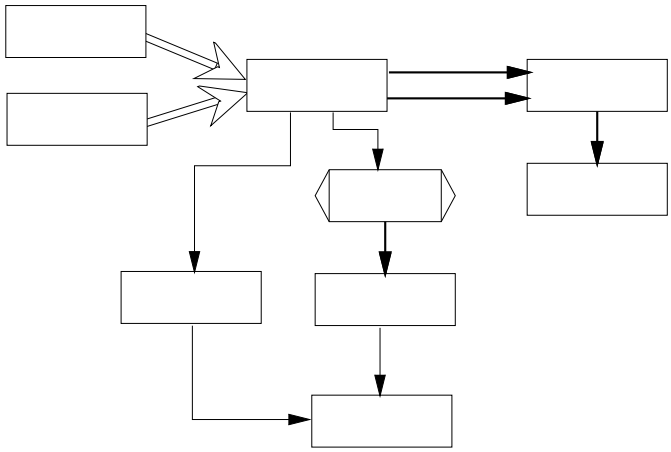

Figure 1: A class dictionary graph of schema `Trip`.

In the object reference model, we describe the structure of objects and classes in terms of a class dictionary graph (or so called schema graph). Two kinds of classes are distinguished: *alternation classes* and *construction classes*. Alternation classes are regarded as abstract classes. Construction classes are instantiable classes. Two kinds of relationships are distinguished between classes: inheritance relationships (called *alternation* or *sepcialization edges*) and object reference relationships (called *construction edges*). We refer to the set of is-a relationships of a schema as the specialization (or is-a) hierarchy and the set of object construction relationships as construction hierarchy. An object construction hierarchy may consist of several disconnected subgraphs with possibly multiple construction edges, and construction loops or self construction loops. Information about what methods need to be attached to a class is deliberately omitted from the class dictionary graph at this stage; it will be "injected" into a class via propagation patterns at method propagation time. (See Section 4.1 for details.)

Definition 1 (class dictionary graph)
An object schema G is defined as a labelled, directed class dictionary graph or simply called schema graph, denoted as $G = (V, L, E)$ where V is a finite set of class vertices with a vertex **Object** *as the* root *class; L is a set of labels, each described by a character string; and $E = EA \cup EC$ is a set of edges where EA is a binary relation on $V \times V$, representing* **is-a** *(***specialization***) edges; and EC is a ternary relation on $V \times V \times L$, representing object* **construction** *edges.*

We also denote a schema graph by $G = (V, L, EA, EC)$ for presentation brevity. The root vertex **Object** is a *system-defined* class which contains all the objects of the database, and has neither incoming specialization edge nor incoming construction edge.

Example 1 Suppose we have a class `Trip` which is described by `departure, arrival` and `location-list`. It has a construction edge labeled as `booking` to class `Hotel` too. We may further divide the class `Trip` into two subclasses `BusinessTrip` and `HolidayTrip`. (See Figure 1.) The class dictionary graph of this example can be described as follows:

```
V ={Trip, HolidayTrip, BusinessTrip, LocationList, Location, Hotel, Ident, Time}.
EA={(HolidayTrip,Trip), (BusinessTrip,Trip)}.
EC={(Trip,Location,location-list), (Trip,Time,departure),(Trip,Time,arrival),
   (Trip,Hotel,booking),(Location,Ident,city), (Hotel,Ident,address)}.
```

Definition 2 (specialization reachable: $\Longrightarrow^*$)
Let $G = (V, L, EA, EC)$ be a schema graph. A class $u \in V$ is specialization reachable from class $v \in V$, denoted by $v \Longrightarrow^* u$, iff one of the following conditions is satisfied:

(i) $u = v$.
(ii) $(v, u) \in EA$.
(iii) $\exists w \in V,\ w \neq u,\ w \neq v$ *s.t.* $v \Longrightarrow^* w,\ w \Longrightarrow^* u$.

Let us take a look at Figure 1. According to Definition 2(ii), class `Trip` is specialization reachable from class `HolidayTrip`. Note that the following anti-symmetric restriction is placed on EA: if $\forall u \in V, \exists v \in V$ s.t. $(u, v) \in EA$, then $(v, u) \notin EA$. In fact, "$\Longrightarrow^*$" is the transitive closure of EA.

Definition 3 (construction reachable: $\longrightarrow^*$)
Let $G = (V, L, EA, EC)$ be a schema graph. A class vertex $u \in V$ is construction reachable from class vertex $v \in V$, denoted by $v \longrightarrow^* u$, iff one of the following conditions is verified:

(i) $\exists \ell \in L$ *s.t.* $(v, u, \ell) \in EC$.
(ii) $\exists w \in V,\ \ell \in L$ *s.t.* $v \Longrightarrow^* w$ *and* $(w, u, \ell) \in EC$.
(iii) $\exists w \in V,\ w \neq u,\ w \neq v$: $v \longrightarrow^* w,\ w \longrightarrow^* u$.

Consider Figure 1. By the condition (ii) and (iii) of this definition, class vertex `Location` is construction reachable from class vertex `HolidayTrip`, because we have `HolidayTrip`$\Longrightarrow^*$`Trip` and `Trip`$\longrightarrow^*$`Location`. This example demonstrates that, by applying both construction reachability and specialization reachablity, we may directly obtain the inheritance property of the *is-a* class hierarchy (cf.[Car84]). More specifically, when a class vertex u is *specialization reachable* from class vertex v, then for any other vertex $w \in V$, if w is construction reachable from u, we may prove that w is also construction reachable from v (cf.[**?**]).

In order to obtain a good understanding of the contributions of the EVOLVE development, before giving an overview of the EVOLVE framework, we first introduce and discuss the two main features of the EVOLVE approiach: the development of a basic set of adaptive schema design rules and the two reuse mechanisms for operational requirement sepcifications respectively. We will also demonstrate by examples on why these mechanisms can be used as adaptive specification techniques for object-oriented software evolution.

3 Adaptive Schema Design

3.1 A quick introduction to the basic set of style rules

In the EVOLVE project, we develop a selection of design and evaluation rules for building an adaptive object-oriented schema. This set of style rules include not only those which we use to preserve validity

and minimality of an object-oriented schema, but also those which help us to promote extensibility, reusability and adaptiveness of an object-oriented schema against future requirement changes.

The style rules for preserving the validity of an object-oriented schema guarantees that it should have **no-dangling class vertex**, and satisfy the **cycle-free specialization** axiom and the **unique construction label** axiom. These are actually the common baselines for correct use of the object-oriented modeling concepts. Once a schema satisfies these three validity rules, we refer to the schema as a valid normal form (valid-NF).

Axiom 1 (no dangling class vertex)
Let $G = (V, L, EA, EC)$ be a schema graph. We say that the schema G has no dangling classes, if there is no class vertex in G, which has neither outgoing edges to other vertices nor incoming edges from other vertices. Two cases need to be distinguished:

(i) $\forall u \in V, \exists v \in V, v \neq u$ *s.t.* $(v, u) \in EA$.
(ii) $\forall u \in V, \exists v \in (V - \{\textbf{Object}\})$, $v \neq u$ *s.t.* $(v, u) \in EA \vee (u, v) \in EA \vee u \in Parts(v) \vee v \in Parts(u)$.

Axiom 2 (cycle-free specialization axiom)
Let $G = (V, L, EA, EC)$ be a class dictionary graph. We say that G has no specialization cycle, if and only if $\forall v \in V, \nexists u \in V$ s.t. $u \neq v, u \Longrightarrow^* v, v \Longrightarrow^* u$.

Axiom 3 (unique label axiom)
Let $G = (V, L, EA, EC)$ be a schema graph. We say that G follows the unique label axiom only if, for any class vertex in G, all its construction edges (either inherited or locally defined) are distinctly labelled. I.e., the following two conditions should be verified:

(i) $\forall v, u, w \in V, \forall \ell, \ell' \in L$: *if* $(u, v, \ell), (u, w, \ell') \in EC$ *and* $(u, v, \ell) \neq (u, w, \ell')$, *then* $\ell \neq \ell'$.
(ii) $\forall v, u, w, v', w' \in V, \forall \ell, \ell' \in L$: *if* $v \neq v', v \Longrightarrow^* u, v' \Longrightarrow^* u, (v, w, \ell), (v', w', \ell') \in EC$, *then* $\ell \neq \ell'$.

Mechanisms for resolution of the violation to the axiom 3 include, for instance, the renaming mechanism for Eiffel and the mechanisms proposed in literature for resolution of name conflicts of multiple inheritance. This axiom has been widely used in many object-oriented models that support subtype inheritance (cf. [BCG+87, LX93]).

The style rules for good use of inheritance concept presents a design guideline to encourage **abstraction of common components** among classes by promoting inheritance along specialization hierarchy, and to advocate **minimization of multiple inheritance** used in a schema graph. Therefore, for any schema, if it meets these two inheritance style rules, we consider it to be of inheritance normal form.

We also develop two style rules for preserving minimality of a schema specification. The first one is called **specialization minimality**. When we say that a schema is semantically minimal or non-redundant, if no concept can be deleted from the schema without losing information. Redundancy in an object-oriented schema may occur for several reasons. One of the most common causes is due to the fact that application requirements often have an inherent redundancy and this redundancy migrates into the initial schema design. Redundancy can also be incurred when related concepts are expressed

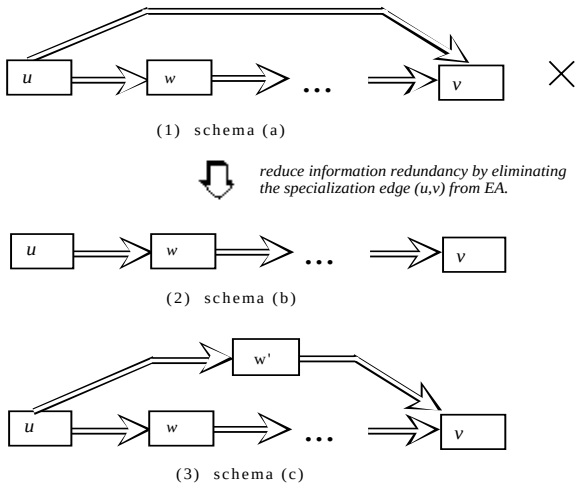

Figure 2: Illustration of the specialization minimality axiom

in different schemas and these schemas need to be merged such as required in the heterogeneous and distributed database environment [SL90]. Furthmore, redundancy of an object-oriented schema is often embedded in several ways within the schema. In the EVOLVE development, we focus on two aspects of the schema redundancy: *specialization redundancy* and *useless classes*. Specialization redundancy exists when a specialization edge between two class vertices has the same information content as a path of the specialization edges between the two classes. We may describe a specialization path from vertex u to vertex v by a list of vertices $< u, w_1, w_2, ..., w_n, v >$ $(n > 1)$, satisfying that (u, w_1), (w_1, w_2), ..., $(w_n, v) \in EA$. The style rule for promoting the specialization minimality of a schema graph is to eliminate all the redundant specialization edges implied in specialization abstraction (see Figure 2(a)(b)). It is worth noting that, however, a schema with more than one specialization path between two classes does not necessarily imply specialization redundancy (see the schema (c) in Figure 2).

The second style rule for minimality is called **no-useless classes**. For any class vertex v, if it has neither incoming specialization edge, nor outgoing construction edges, then it is desirable and also reasonable to consider vertex v as a useless class, because it contains no construction information at all. Most of the useless class specifications initially come from application requirements, and mainly caused by the difficulty in distinguishing between what should be modelled as classes and what are properties of a class in the logical design. Whenever a schema graph preserves specialization minimality and has no useless classes, we consider it satisfying the minimality normal form.

We have also defined the normality rule for eliminating the update anomalies implied in the initial schema design, as well as the cycle-free dependency rule for minimizing the existence dependency loops implied in the construction hierarchy of the initial schema design.

In the next subsection, we formally introduce the inheritance style rules and the concept of inheritance normal form. The benefits of using such style rules are illustrated through examples.

3.2 Style rules for enhancing the extensibility of an object-oriented schema

Extensibility of a database schema is defined by the adaptiveness and the flexibility of the schema in anticipation of future schema changes. The crucial factor for extensibility is to reduce the impact of a schema modification (i)on the entire structural consistency of the existing schema (cf. [BKKK87, Zic91], (ii)on the organization of the databases (cf.[LH90]), and (iii)on the workload required for rewriting of the existing application programs (cf. [SZ87]). The basic principle for achieving better extensibility in an object-oriented database system is to advocate minimal coupling between abstractions (e.g., methods, procedures, and modules) and to provide a certain degree of information hiding and information restriction in order to reduce the cost of software changes. In what follows, we concentrate on how to make good use of inheritance feature along with specialization hierarchy.

3.2.1 Information Localization

The following axiom presents a design guideline to encourage abstraction of common components among classes by promoting inheritance along specialization hierarchy.

Axiom 4 (information localization)
Let $G = (V, L, EA, EC)$ be a schema graph. We say that G supports information localization if the following condition is verified: $\forall u, v, w \in V$: if $u \neq v$, then $\nexists \ell \in L$ s.t. $(u, w, l), (v, w, l) \in EC$.

In order to satisfy this axiom, an object-oriented schema should abstract all the common components among the constructed class vertices by introducing inheritance through addition of specialization edges. The advantage of encouraging information localization through abstraction of common parts is apparent. Let's look at the following example.

Example 2 Consider the two schema graphs given in Figure 3. In the schema graph (a), the two class vertices u and v have a common class vertex w via the two construction edges with the same label ℓ. Obviously it violates Axiom 4. As a consequence, whenever a change occurs to either of these construction edges or to the class w (for instance renaming of the label ℓ or of the class name w), both class u and v will be affected accordingly. Furthermore, such a change may cause a need for modification of the existing software programs (e.g., database queries) that are working with the class u and v.

In order to reduce the possible impact of a schema modification to the database and application programs working with the schema (a), we need to promote information localization by abstracting the common information among the class u and v. For example, we may introduce a new class vertex w', connecting w' to vertex w by building a construction edge with the same label ℓ, and meanwhile add two specialization edges from class u and v to the new class w'. Thus, the class w can still be construction reachable from both class u and v with the same label ℓ (see the schema graph (b) in Figure 3). The only difference between these two schemas is that, in contrast to the schema (a), the class w in the schema (b) becomes construction reachable from class u and v through a combination of specialization abstraction with construction abstraction, rather than via direct construction edges. Clearly, the schema graph (b) satisfies Axiom 4.

Now if we want a change, for instance, by renaming the label ℓ of the construction edge (w', w, ℓ), we only need to carry out the change at one place (i.e., in the class w'). Furthermore, such change has

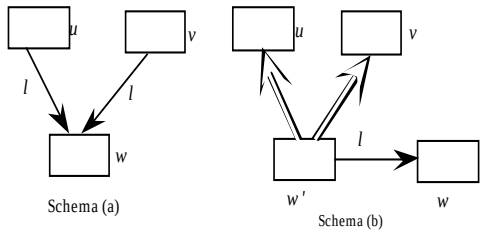

Figure 3: Examples for illustration of Axiom 4

no modification impact on any of the existing programs working with the classes u and v, because the change can automatically be reflected to class u and v through inheritance implied by their specialization edges to vertex w'. Besides, thank to the fact that the transformation of the schema graph (a) into the schema graph (b) is actually object-preserving (cf.[Ber91]) all the programs previously working with the schema(a) are still applicable to the optimized schema(b).

3.2.2 Minimization of Multiple Inheritance

We below present an axiom for minimizing the amount of multiple inheritance used in a schema graph. The reason is simply because multiple inheritance causes extra maintenance overhead in comparison with single inheritance. To achieve better extensibility and maintainability, a schema should minimize multiple inheritance as much as possible. The concept of total generalization plays an important role in obtaining this objective.

Definition 4 (total generalization)
Given a schema graph $G = (V, L, EA, EC)$, we say that a class vertex u is a **total generalization** *of class vertices v_1, ..., $v_n \in V$ $(n \geq 1)$, if and only if, for (u, v_1), ..., $(u, v_n) \in EA$, the following two conditions are verified:*

(i) $\nexists w \in V$: $w \neq v_1$, ..., $w \neq v_n$, $(w, u) \in EA$;
(ii) **content**(u) = **content**$(v_1) \cup$ ... $\cup$ **content**(v_n).

We define the set of outgoing specialization edges from class vertex u, denoted as $OutEA(u)$, as follows: for any class vertices u, v_1, ..., $v_n \in V$, if the class vertex u is a total generalization of the classes v_1, ..., v_n, then $OutEA(u) =_{def} \{v_i \mid v_i \in V, (v_i, u) \in EA\}$. Otherwise $OutEA(u) =_{def} \emptyset$.

Axiom 5 (multiple inheritance minimization)
A schema graph $G = (V, L, EA, EC)$ is said to support multiple inheritance minimization, if the following rules hold.

(i) Complete cover
$\nexists u, v \in V$ *s.t.* $OutEA(u) \neq \emptyset$, $OutEA(v) \neq \emptyset$, *and* $OutEA(v) \subset OutEA(u)$.
(ii) Partial cover
$\nexists u, v \in V$ *s.t.* $OutEA(u) \neq \emptyset$, $OutEA(v) \neq \emptyset$, *and* $\mid OutEA(u) \cap OutEA(v) \mid > 1$; *where* $\mid OutEA(u) \cap OutEA(v) \mid$ *denotes the total number of the elements in the set* $(OutEA(u) \cap OutEA(v))$.

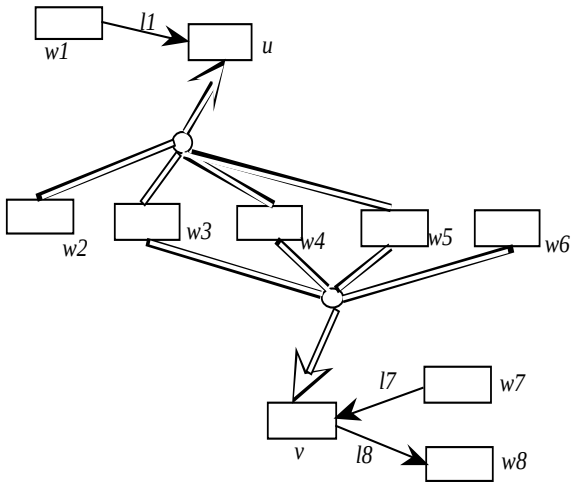

Figure 4: An example schema that violates Axiom 5

This Axiom suggests that an adaptive schema should always provide a support for minimization of multiple inheritance. More specifically, the complete cover rule requires that whenever a schema graph has a total generalization (say $OutEA(v)$) which is completely covered by another total generalization (say $OutEA(u)$), i.e., $OutEA(v) \subseteq OutEA(u)$, it would be helpful to reorganise them, for example, by adding a new specialization edge from u to v so that those specialization edges anchored at both u and v can be replaced by only to be anchored at the class v. The partial cover rule of Axiom 5 amounts to saying that if two classes have outgoing specialization egdes to more than one common class vertex, then it is desirable to introduce a new class vertex to localize the common information in order to increase adaptiveness of a schema graph against future schema changes.

Example 3 Consider the schema graph in Figure 4. Assume that class u is not specialization reachable from class v and vice versa. The class u is a total generalization of w_2, w_3, w_4, w_5. The class v is a total generalization of w_3, w_4, w_5, w_6. Thus we have $OutEA(u) = \{w_2, w_3, w_4, w_5\}$ and $OutEA(v) = \{w_3, w_4, w_5, w_6\}$. $OutEA(u) \cap OutEA(v) = \{w_3, w_4, w_5\}$, and $\mid OutEA(u) \cap OutEA(v) \mid = 3 > 1$. Clearly, this schema graph violates the partial cover rule of Axiom 5. Problems may occur whenever we require a change to the information that is common to classes w_3, w_4, w_5, because such changes have to be carried out by repeating the same modification to each of the aforementioned classes w_3, w_4, w_5. We may solve the problems as such by transforming the schema graph in Figure 4 into an equivalent schema graph as shown in Figure 5.

This transformation can be fulfilled by introducing a new class vertex w and creating outgoing specialization edges from class w to class u and v, and meanwhile replacing those specialization edges from class w_3, w_4, w_5 to both u and v, by adding specialization edges from class w_3, w_4, w_5 to w. Clearly, the optimized schema graph in Figure 5 satisfies Axiom 5. The benefit of such transformation is obvious. For instance, concerning the schema in Figure 5, any change which is common to the class w_3, w_4, w_5, now only need to be performed at one place (i.e., in the newly added class w). The change will automatically be reflected to the class w_3, w_4, w_5 through specialization inheritance (cf. [Car84]).

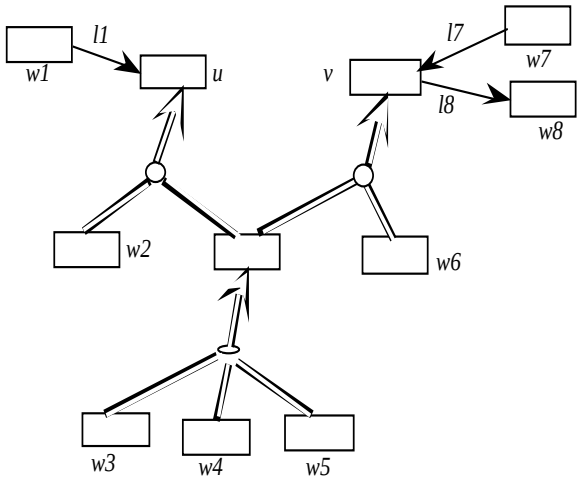

Figure 5: A modified schema graph that satisfies Axiom 5

Definition 5 (Inheritance Norm Form: I-NF)
Let $G = (V, L, EA, EC)$ be any schema graph and I-NF denote a collection of adaptive schema graphs by making the best use of inheritance feature. A schema graph G belongs to I-NF, if and only if (i)G belongs to V-NF and (ii)G satisfies Axiom 4 and Axiom 5.

3.3 Remarks

The basic set of the EVOLVE style rules forms a framework for axiomatic characterizations of object-oriented schemas in terms of adaptiveness and robustness of an object-oriented design. This framework consists of a selection of normal forms, i.e., valid NF, inheritance NF, minimal NF, normalized NF and cycle-free dependency NF (D-NF). So far, we have informally overviewed the basic set of adaptive schema style rules. We have also presented the formal development of the style rules for valid normal form and for inheritance normal form. However, due to the space limitation, in the current presentation, we omit the formal definition for the minimality normal form, the normalized normal form as well as the cycle-free dependency normal form. Readers who are interested in may refer to the technical report [**?**] for further details.

In the EVOLVE development, we regard the valid NF as the baseline for every object-oriented schema design since it exhibits the basic assumption of object-oriented modeling. It is interesting to note that a schema which belongs to, for example, inheritance-NF, may not be necessarily in the other normal forms (e.g., minimal NF, normalized NF, or cycle-free dependency NF). By using this framework, many desired properties of a schema design, such as validity, extensibility, minimality, normality, etc., can be well understood under the context of object-oriented models. For example, a inheritance-NF schema promotes maximum information localization so that the impact of schema changes can largely be reduced, a minimality-NF schema encourages that every aspect of the requirements should only appear once in the schema in order to enhance the maintainability of the schema, while a normalized-NF schema avoids update anomalies and therefore when a change happens to the database, the cost for database consistency maintenance is minimized. When a schema belongs to both inheritance-NF and

minimality-NF, it is certainly more adaptive and robust when changes occur to either specialization or construction structure of the schema. We evaluate those schemas that satisfy all the normal forms to be of the mostly higher quality in terms of adaptiveness and robustness of a schema in reacting to the future requirement changes (both at the schema level and at the instance level).

4 Reuse Mechanisms for Adaptive Behavioral Specifications

4.1 Propagation Patterns

The concept of propagation patterns was originally introduced in the Demeter system^{TM} [LXSL91, Lie94] in order to specify object-oriented programs at a higher level of abstraction. We believe that propagation patterns are also useful conceptual programming technique for database applications, which enables system designers and programmers to conceptualize application programs and system behavior with minimal knowledge of the data structure. Put differently, propagation patterns can be seen as a kind of behavioral abstraction of application programs, which defines patterns of operation propagation by reasoning about the behavioral dependencies among cooperating objects. They have proved to be an effective aid for building highly adaptive database programs (methods and queries) and for supporting incremental schema evolution.

4.1.1 A Quick Look at the Syntax

The following information is necessary for the specification of a propagation pattern over a given class dictionary graph, say $G = (V, L, EC, EA)$.

1. *method interface*
   It is described by the method name, the output type (optional) and the set of parameters. The syntax is as follows:
   `OPERATION` $u\ mn(p_1, ..., p_n)$,
   where $u \in V \cup \{void\}$ is either a class vertex in V or the keywork "*void*", indicating an empty result type, $mn \in L$ is the method name and $p_1, ..., p_n$ $(n \geq 0)$, are a list of parameters of this method.

2. *propagation directives*
   Each propagation directive specifies a propagation path and is described by a set of source vertices, a set of propagation constraints and a set of target vertices.

   - *a set of source vertices*
     specifies where the method propagation pattern starts. It is a mandatory information of a propagation pattern. The syntax is
     `FROM` $u_1, u_2, ..., u_s$,
     where $u_1, u_2, ..., u_s$ $(s \geq 1)$ are class vertices in the given class dictionary graph.
   - *propagation constraints*
     indicate the restrictions over the propagation pattern. They are described by a set of propagation restriction or exclusion constraints. The former specifies which edges have to be passed through along the road of the method propagation and is described by the following syntax:

THROUGH $e_{i_1}, e_{i_2}, ..., e_{i_p}$.

The latter identifies which edges should be excluded from the specified method propagation path. The syntax specification is given below.
BYPASSING $e_{j_1}, e_{j_2}, ..., e_{j_q}$.
Note that $e_{i_1}, e_{i_2}, ..., e_{i_p}$ $(p \geq 0)$ and $e_{j_1}, e_{j_2}, ..., e_{j_q}$ $(q \geq 0)$ are edges in the given class dictionary graph.

- *a set of target vertices*
  specifies with which classes the propagation pattern terminates. It is also a mandatory component of a propagation pattern specification. The following is the syntax description:
  TO $w_1, w_2, ..., w_t$,
  where $w_1, w_2, ..., w_t$ $(t \geq 1)$ are class vertices in the given class dictionary graph.

3. *method annotations*
   A method annotation consists of a sequence of code fragments. Each code fragment is described by a class vertex with the code enclosed within "@" symbols. Three types of method annotations are currently provided: PREFIX (which specifies a prefix to the traversal code), SUFFIX (which serves as a suffix to the traversal code) and PRIMARY (which replaces the traversal code). The specification syntax of the method annotations are the following:

   PREFIX u_1 (@ f_1 @),..., u_m (@ f_m @)
   SUFFIX v_1 (@ g_1 @),..., v_s (@ g_s @)
   PRIMARY w_1 (@ h_1 @),..., w_t (@ h_t @)

   where $u_i, v_j, w_k \in V$ are class vertices, f_i, g_j, h_k are code fragments (e.g., C++ code fragments), $1 \leq i \leq m$, $1 \leq j \leq s$, $1 \leq k \leq t$.

4.1.2 Illustration By Examples

To illustrate the syntax and semantics of propagation patterns, we provide some representative examples in this section and omit the formal definition and formal analysis of propagation patterns. Readers who are interested in the formal treatment may refer to [**?**].

Example 4 Consider the class dictionary graph as shown in Figure 1. The `Trip` objects have parts called `departure` and `arrival` which can be printed. A `Trip` object contains a list of `Location` objects, each has an `Ident` object as a component, describing the city to be visited during the trip. Suppose we want to have a method "*print-itinerary*" which prints only the `departure` time and the list of cities to be visited, followed by the `arrival` time. All the `Hotel` objects of a trip should be excluded from this printing task. We may define the method "*print-itinerary*" by writting a propagation pattern (see Figure 6). The idea behind this propagation patterns is based on the fact that a number of classes in the class dictionary of Figure 1 need to cooperate to accomplish the task "*print-itinerary*", but only a few information are necessary for specifying this task, since the rest can easily be derived from the structural specifications of the schema. For instance, in Figure 6, we specify the interface of the method to be propagated with the clause `OPERATION void print-itinerary`. The source of this propagation pattern is given with the clause `FROM Trip`, specifying where the propagation pattern starts. The target of this propagation pattern is provided with the clause `TO Ident`, indicating with which class(es) the propagation pattern terminates. The clause `BYPASSING *,booking,*` identifies the restriction (propagation constraints) over this propagation pattern in order to exclude all the hotel

```
OPERATION void print-itinerary()
FROM Trip
BYPASSING *,booking,*
TO Ident
PREFIX Trip (@ departure -> g_print() @)
SUFFIX Trip (@ arrival -> g_print() @)
PRIMARY Ident (@ this -> g_print() @).
```

Figure 6: An example propagation pattern *print-itinerary*.

information from this particular printing task. The clauses `PREFIX Trip`, `SUFFIX Trip` and `PRIMARY Ident`, followed by the actual programming code (e.g., C++ code) surrounded by "(@" and "@)", specify the method body. We call the source clause, the target clause and the propagation constraint clause together the *propagation directive* of pattern "*print-itinerary*".

Remarks:
(i)Writing a propagation pattern does not require knowledge of the detailed data structure. One obvious benefit of this feature is to allow reuse of propagation patterns for several similar data structures and thus to increase the adaptiveness of an object-oriented schema design.

(ii)If the `BYPASSING` option is not included in the above propagation pattern, instead, only the clause `FROM Trip` and clause `TO Ident` had been used, then the edge (`Trip,booking,Hotel`) would have participated in the propagation too. It means that the propagation path implied by the given propagation directive will include both the path from `Trip` through `Location` to `Ident` and the path from `Trip` through `Hotel` to `Ident`. It means that the task "*print-itinerary*" will print both the departure time, the list of locations to be visited, the hotel booked for the trip, and the arrival time.

(iii) Propagation patterns can automatically or semi-automatically be translated at method propagation time into any object-oriented programming language code, for example C++ code, and meanwhile the corresponding code fragments will be inserted into the appropriate classes which participate in the propagation pattern traversal.

To illustrate the remark (iii), let's consider a simplied `Trip` schema given in Figure 7 for presentation convenience. We may therefore model the operational requirement described in Example 4 by means of the propagation pattern defined in Figure 8. In this example, the prefix annotation is used to print the `departure` time before the `Trip` object is traversed; the suffix annotation prints the `arrival` time after the `Trip` object has been traversed. The primary annotation replaces the default traversal code when printing the current `Ident` object. When the above propagation pattern is injected into the class structure given in Figure 7 at propagation time, program fragments will automatically be generated according to the given method annotations. (See the C++ codes attached to the classes in Figure 7, which we obtained by running the EVOLVE-Demeter/C++ on the example.) Note that the class `LocationList` in Figure 7 is introduced here merely as a list construct for showing how the code is generated for implementing the requirement that a `Trip` consists of a list of `Location` objects. The C++ method definitions attached to each class in Figure 7 is generated according to the propagation pattern given in Figure 8. The completeness of these C++ methods fully depends on the specification details of the propagation patterns.

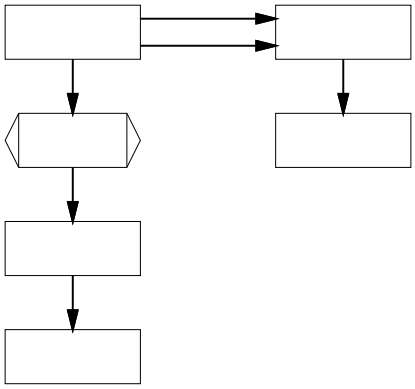

Figure 7: The `Trip` schema with generated C++ code attached to corresponding classes.

```
OPERATION void print-itinerary()
FROM Trip
TO Ident
PREFIX Trip (@ departure -> g_print() @)
SUFFIX Trip (@ arrival -> g_print() @)
PRIMARY Ident (@ this -> g_print() @);
```

Figure 8: The propagation pattern *print-itinerary* over the `Trip` schema in Figure **??**.

Clearly, by using propagation patterns, any unnecessary information about the class structure need not be hardwired into the specification. This allows the specification of a propagation pattern to be more flexible towards schema modification. For example, suppose the schema shown in Figure 7 is extended by adding a new class `DayTrip` such that a `Trip` object now contains a list of `DayTrip` objects and each `DayTrip` object contains a list of `Location` objects which are printable through `Ident` objects (see the schema presented in Figure 9). Although the propagation pattern in Figure 8 is defined over the schema in Figure 7, the modification on the `Trip` schema requires no reprogramming of this method, because all the key information (hooks) of "*print-itinerary*" (such as `Trip,departure,arrival,Ident`) are included in the modified schema (See Figure 9). It means that this propagation pattern is also applicable to the schema of Figure 9. We can actually reuse the propagation pattern defined in Figure 8 for the `Trip` schema in Figure 9 without changing the specification of the propagation pattern and thus the code generated previously based on this propagation pattern.

To further demonstrate the robustness of database programs written in propagation patterns, let's use the conventional object-oriented database languages to express the operation "*print-itinerary*", and show how it reacts to the above schema modification. Using most of the existing object-oriented languages, the method designers or query writers must query the objects of interest by their precise

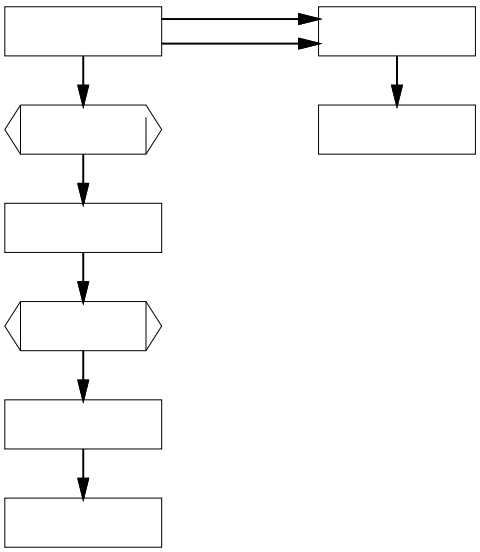

Figure 9: A second `Trip` schema with generated C++ code.

path expressions. For example, the "*print-itinerary*" has to be expressed more or less as follows:

PRINT *t.departure, t.locations.city, t.arrival*
FROM *t* in `Trip`.

Whenever we modify the `Trip` schema shown in Figure 7, for example, by simply adding a new class `DayTrip` as presented in Figure 9, then the above SQL-like operation expression is not anymore a valid operation to the newly modified schema (Figure 9). We must rewrite the above operation expression manually in order to replace the "old" path expression "*t.locations.city*" by "*t.daytrips.locations.city*". Imagine that if we have a number of application programs implemented over the "old" `Trip` schema (see Figure 7), one single modification on this schema could cause possibly rewriting of all the existing application programs which have been working with the schema. This is by no means a pain. However, if, instead, we use propagation patterns to specify this "*print-itinerary*" operation (See Figure 8), the modification as such will have no impact on the "existing (old)" database programs; because writing the propagation pattern "*print-itinerary*" requires neither the detailed structure of `Trip` schema nor the navigational information of how to traverse `Ident` from `Trip`. The propagation pattern "*print-itinerary*" defined in Figure 8 may remain unchange and works as a valid method on the modified `Trip` schema. The main idea behind propagation patterns is simply to delay the binding of the concrete propagation paths used in each method or query specification from method (or query) writting time to operation propagation time, prior to compile time.

In our view, propagation patterns, on the one hand, proposes a novel method specification technique for promoting adaptive object-oriented schema design; and on the other hands, can be used as a database programming language for enhancing robustness of database programs.

4.1.3 Polymorphic Character of Propagation Patterns

As stated earlier, by using propagation patterns to model the dynamic part of an object-oriented database system, we may achieve a certain degree of adaptiveness and flexibility of the database specifications against future changes, especially with respect to several types of structural changes. For example, the propagation pattern given in Figure 8 is defined over the class dictionary graph of Figure 7, but it can also be used as a propagation pattern for the class dictionary graph given in Figure 9, because both class schemas include `Trip` objects, which have `Ident` objects as parts. More interestingly, this propagation pattern is actually applicable to a family of `Trip` class structures, as long as the `Trip` class has a `departure` and an `arrival` part, and a "path" to class `Ident`. In short, by using propagation patterns, schema designers and programmers can focus on merely the most interesting components of the class structure. No precise knowlegde about how the structural details are modeled in a particular schema (class dictionary graph) is required. We call this particular feature the polymorphic character of propagation patterns.

It is interesting to note that, for the same pattern *print-itinerary*, its propagation scope over the class schema of Figure 7 is not the same as the one over the class schema of Figure 9. Thus, the binding of this propagation pattern to the involved classes in the schema of Figure 7 is different from the one in Figure 9.

Generally speaking, the polymorphism of propagation patterns belongs to the family of ad-hoc polymorphism [CW85]. We believe that the theory of polymorphism may provide a sound theoretical basis for investigating the adaptiveness of object-oriented schema design and schema evolution in both the structural and the dynamic aspect.

4.2 Behavioral Refinement of Propagation Patterns

Up to now, we have shown by examples that propagation patterns are a promising conceptual programming technique for modeling and programming the dynamic behavior of object-oriented database systems, because of their adaptiveness to the structural changes of a schema.

The adaptiveness of propagation patterns results from a number of interesting features.

- First of all, the specification of propagation patterns does not require hard-wiring them to a particular class structure. This leaves room for deriving behavioral abstraction based on structure abstraction and for incremental design of methods (e.g., propagation patterns).
- Secondly, propagation patterns are defined in terms of only a few, essential classes and relationship specifications. They serve as hooks into the class structure [Lie94]. The rest of the knowledge required for behavior implemention can actually be derived on the basis of these hooks and the corresponding class schema.
- Thirdly, propagation patterns promote the well-known concept of late binding. Instead of binding methods to classes at program-writing time, propagation patterns encourage the binding of methods to classes at propagation time, prior to compile time. Therefore, given a propagation pattern defined over a class structure (say G), any change to the structure of G (which does not affect the hooks of this propagation pattern) will have little impact on the specification of this propagation pattern, even though its scope over the modified class schema could be changed accordingly. In other words, the given propagation pattern specification by itself can be reused in

a modified class schema, if no additional propagation constraints or method annotations are required. But the binding of the method interface and annotations to the relevant classes may need to be re-adjusted implicitly at propagation time (through propagation pattern interpretation).

In contrast, when changes are required to the dynamic (behavioral) aspect of a schema and thus to some existing propagation patterns, it becomes indispensible to redefine the affected propagation patterns or to further extend some existing propagation patterns. It is definitely beneficial if some reuse mechanisms are provided so that the adaptation of existing propagation patterns to the new requirement changes do not have to start from scratch or be rewritten completely, even if the affected propagation patterns are simple ones. Because once a propagation pattern is reused, both the programing codes generated in terms of it and the existing binding of methods to classes at propagation time may inherently be reused as well. Besides, by reusing the specification of propagation patterns, information involved is maximumly localized such that any change to the existing specifications is carried out only at one place. The effort to manually preserve the consistency of the specifications due to schema modification is then minimized. We call the mechanism as such the behavioral refinement mechanism for propagation patterns.

Example 5 Consider the `Trip` schema in Figure 9 and the propagation pattern for printing trip itineraries in a travel agency defined in Figure 8. Suppose now we want to modify the `Trip` schema of Figure 9 by adding a new property `Date` to class `DayTrip`. We also want to extend the task of printing trip itineraries by adding a new operational requirement that, for each trip, the `date` for every travel day must also be printed. Comparing with the "old" propagation pattern "*print itinerary*" defined in Figure 8, this extended task (let us call it "*print-detailed-itinerary*") obviously includes all the functionalities of the "old" propagation pattern "*print-itinerary*" (see Figure 8) and also some additional propagation constraints and method annotations. For instance, the following annotation needs to be added for printing the date of each travel day within a trip:

```
WRAPPER DayTrip
PREFIX (@ date -> g-print() @)
```

Additionally, in the schema of Figure **??**, there is more than one path from `Trip` to `Ident`: one through the edge *locations* and the other through the edge *date*. We also need to add the following extra propagation constraint into the propagation pattern defined in Figure 8:

```
THROUGH (DayTrip,locations,LocationList)
or
BYPASSING (DayTrip,date,Date).
```

There are two ways to accomplish this operational requirement change. One way is to redefine (rewrite) the previous propagation pattern "*print-itinerary*" completely and then redo the binding (injection) of methods to classes at propagation time. For example, we rewrite the propagation pattern "*print-itinerary*" completely, even though most of the previous bindings will remain the same for this redefined propagation pattern (such as the C++ code for class `Trip, DayTripList, LocationList, Location, Ident` will remain exactly the same). The other way is to employ some reuse mechanisms so that more specialized propagation patterns can be defined in terms of existing ones. This means that only the propagation constraint and the method annotation which are new need to be defined. The rest can directly be reused from (or shared with) the existing pattern by means of the propagation refinement mechanism. Furthermore, at the propagation time of the refined propagation pattern, only the new method annotations need to be injected to the involved classes, since all previous bindings and code

```
OPERATION void print-detailed-itinerary()
BEHAVIORAL REFINEMENT OF Print-itinerary
ADD CONSTRAINT THROUGH (DayTrip,locations,LocationList)
ADD ANNOTATION
   WRAPPER DayTrip
   PREFIX (@ date -> g-print() @).
```

Figure 10: A refined propagation pattern of "*print-itinerary*".

generated in terms of the "old" method annotations may be reused accordingly. For example, to represent the updated printing task, we may reuse the propagation pattern defined in Figure 8, because only one prefix annotation and one extra propagation constraint need to be added. Moreover, the addition of `date` into the task of printing trip itineraries, in fact, only affect the "old" binding of the method ("*print-itinerary*") to the class vertex `DayTrip` and the code generated for this binding. The rest remains exactly the same. Thus, by using the propagation refinement mechanism, we may specify the desired requirement change as shown in Figure 10.

Generally speaking, the refinement of propagation patterns is a behavioral abstraction mechanism, which allows us to define more specialized propagation patterns in terms of existing propagation patterns by restricting propagation behavior to one or more specialized classes as arguments of the method, by imposing extra propagation constraints, or by adding additional method annotations. The idea of providing a support for the propagation pattern refinement is to enable more specialized propagation patterns to be defined in terms of existing ones such that only the propagation constraint and the method annotation which are new need to be defined. The rest can be directly reused from (or shared with) the existing pattern by means of the propagation refinement mechanism. Furthermore, at propagation time of the refined propagation pattern, only the new method annotations need to be injected to the involved classes, since all previous bindings and code generated in terms of the "old" method annotations may be reused accordingly. Due to the space limitation, we omit the formal definition of propagation pattern refinement in this short paper. Readers may refer to [**?**] for more details.

In summary, the propagation pattern refinement mechanism helps to increase the flexibility and adaptiveness of propagation patterns against future operational requirement changes. It can also be useful for promoting the concept of propagation pattern inheritance under a class dictionary graph. Another interesting feature of propagation pattern refinement is presented by its transitivity. We have formally proved that for any propagation pattern α, β, γ, if pattern α is a behavioral refinement of pattern β and pattern β is a behavioral refinement of pattern γ, then pattern α is also a behavioral refinement of pattern γ (see [**?**]).

5 Overview of the EVOLVE project

5.1 The Framework

The framework of the EVOLVE development consists of two layers:

- The Design and Prototyping Layer.
- The C++ Code Generation Layer.

The Design and Prototyping Layer first translate the given requirement specifications (including both structural and behavioral requirements), that are produced through requirement capturing and specification process, into the EVOLVE notation, i.e., modeling the structural specifications in terms of the class dictionary graph notation (see Section 2), and representing the operational sepcifications in terms of the propagation patterns (see Section 4). Since the EVOLVE development takes the Demeter system/C++ as the experiment base, most of the EVOLVE notation are compatible with the Demeter notation [Lie94]. After the syntax and semantic checking of the class dictionary specification and the propagation pattern schema, The EVOLVE will analyze and measure the quality of a given schema design by means of the set of adaptive schema style rules (see Section 3). Such an analysis is based on enhancing the adaptiveness and robustness of the schema in anticipation of future requirement changes. Based on the analysis results produced through the schema analysis and evaluation stage, the EVOLVE will use its schema transformation algorithms to generate a better style schema for the initial requirement design and specifications. The modified schema presents relatively higher adaptiveness towards future requirement changes. A list of benefits may also be provided, if requested, for demonstrating why and in what aspects the modified schema is better in terms of adaptiveness and robustness of the schema design.

In the current implementation scenario, the Code Generation layer partially uses the Demeter system/C++ to assist in generation of the corresponding C++ class library and C++ method definitions for a given class dictionary graph and the set of propagation patterns ranging over this class dictionary graph. We are also interested in experiment of mapping the generated C++ code to a C++ based object schema for a commercial object-oriented DBMS (e.g., OpenDB, ObjectStroe, O2, etc.).

5.2 Outline of the Methodology

We describe the methodology of the EVOLVE development in Figure 11. It consists of four components: the schema editor, the adaptive schema style analyzer, the schema optimizer and the code generator.

- The EVOLVE object-oriented schema editor is composed of two components: class dictionary builder and behavioral specification editor. The former is designed to transform, for example, an ER-like modeling sepcification into an object-oriented class dictionary graph. It is also possible to model the structural/organizational requirements of a universe of discourse directly through the EVOLVE graphic modeling tool. The latter intends to translate the given operational sepcifications into the EVOLVE method definitions in terms of propagation patterns and the refinement mechanism of propagation patterns.
- The adaptive schema style checker is also called schema analyzer. It applies a set of the EVOLVE adaptive schema style rules to the given class dictionary graph and results in a list of advice for schema modification, a proposal for transforming the initial schema design into a better style schema in terms of adaptiveness, and possibly a benefit report if requested.
- Upon the agreement with the schema designers, the adaptive schema optimizer will transform the original schema design into an "optimal" schema (class dictionary graph) which presents better

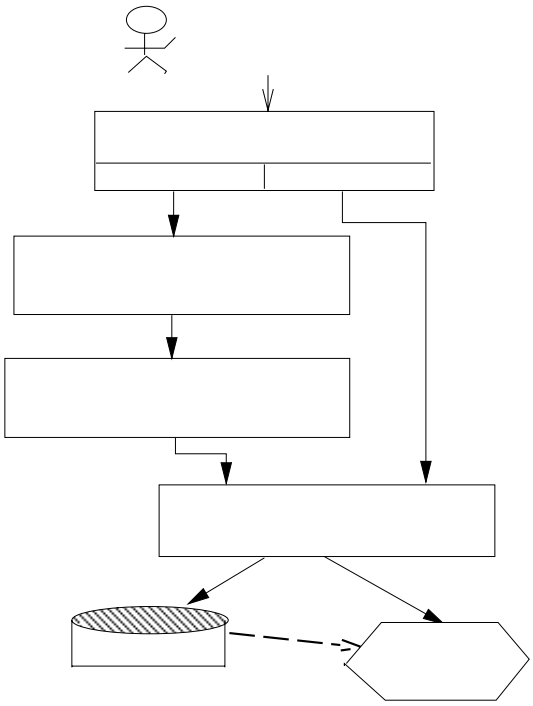

Figure 11: The EVOLVE methodology and toolset

adaptiveness in anticipation of future requirement changes. A key feature of the schema optimization methods is to preserve the information capacity of the original design. For example, the EVOLVE supports two kinds of schema transformation: object-preserving and object-augmenting. The object-preserving transformation guarantees that both the original and the optimized schema define the same set of instantiable classes with the same set of (direct or inherited) properties. While the object-augmenting transformation causes no information loss although it may change the structure of some objects by addition of extra parts or additional classes. More importantly, the distinction of object-preserving and object-augmenting transformations allows us to enhance the reusability of the schema transformation primitives and to facilitate the correctness proof of a given transformation.

- The code generator is developed based on the Demeter System/C++. It consists of two main components. One is to map the optimized class dictionary graph into C++ class dictionary and to generate header files to be associated with each class. The other is to translate the methods written in propagation patterns into C++ application specific method definitions. The Demeter's automatic (or semi-automatic) generation of C++ application specific method definitions is, to our knowledge, rather unique as opposed to several existing CASE tools that only generate C++ "static" code (class library and a set of standard member functions). We are currently also interested in mapping the C++ result to a commercial (preferably C++ based) object-oriented database schema.

6 Concluding Remarks

The increased complexity of object-oriented models necessitates the enhancement of adaptiveness and robustness of an object-oriented design towards changing requirements. The understanding of what properties are critical for construction of an adaptive schema design becomes increasingly important in software evolution and evaluation. In this paper we have presented two groups of techniques for improving the adaptiveness of an object-oriented design and specification:

- The first group of techniques consists of a selection of adaptive schema style rules, such as the style rules for achieving validity, minimality, extensibility, normality of a schema design, and for minimization of dependency loops. We encourage to use this set of style rules proposed as a means for validating quality of a schema, and for transforming an object-oriented schema into a better style in terms of adaptiveness and robustness of a schema design, rather than as a user-oriented method solely for designing the schema. We believe that the set of style rules developed in the EVOLVE project presents an interesting step towards overcoming the potential problems hidden in many object-oriented schema designs.

  Of course, the set of style rules we have introduced is not an exhaustive list. Many issues are still pending better solutions. For instance, it is interesting to study the interactions, among the given style rules and their reasoning capability. We would like to stress that our work on adaptive schema style rules proposes a design framework but not a new model or system. It is targeted towards advanced applications which require the support for more sophisticated object structures and relationships than traditional database application domains. It is, of course, not expected that the proposed set of style rules be incorporated in a given system in its entirty. Rather, the designers may select a useful subset of the style rules, given a particular application domain and data modeling requirement.

- The second group of techniques includes the use of propagation patterns and the mechanism for propagation pattern refinement. The main benefits of using these two techniques are the following.

  - The concept of propagation patterns presents a promising technique for enhancing the robustness of methods and query programs with respect to schema modifications. Using propagation patterns provides method designers and query writers with an opportunity to specify operations without requiring the precise knowledge of the detailed navigational information. Compared with most of the existing object-oriented languages, using propagation patterns, the effort required for manually reprogramming methods and queries due to schema modifications is largely avoided or minimized.
  - The concept of propagation pattern refinement represents an important mechanism for the abstraction and the reuse of propagation patterns. It promotes incremental design of methods and is especially useful for dealing with a class of operational requirement changes. To our knowledge, none of the existing object-oriented specification languages provides a similar support for the incremental definition of methods.

  In comparison with the contract model [HHG90], both propagation patterns and contracts encourage a separation of object behavior specification from object structure specification and both present interesting techniques for operational specification. But there are also a number of differences. First of all, propagation patterns provide better adaptiveness towards schema evolution and change management, because by means of propagation patterns and the propagation pattern

refinement mechanism, the reprogramming of methods and queries due to schema modifications can be avoided or minimized. Second, propagation patterns concentrate more on the specification of and reasoning about operation propagations among a group of related classes; whereas contracts emphasize more on the obligation specification of each participant class in accomplishing a task defined by a group of cooperating classes. Thirdly, the conformance of contracts with classes is required explicitly in the contract model, whereas the conformance of propagation patterns with classes is derived implicitly at propagation time.

It is also interesting to compare the reuse mechanism of propagation patterns with the behavioral abstraction mechanisms defined in the activity model [LM92]. Although there is a similarity between the concept of propagation pattern refinement and the concept of activity specialization, the emphasis and functionality of the activity model is on the declarative specification and reasoning of communication behavior of objects. There is no consideration on specifying and reasoning about operation propagations among cooperating classes.

The recent work presented in [HLM93] has formally studied a number of extension relations of an object-oriented schema based on capturing a similarity at the class structuring level. Such extension relations are useful means for quality control of schema transformations. [BH93] has studied issues on how to preserve or transform propagation patterns under the extended class structures. The work is mainly based on the extension relations identified in [HLM93]. Quite differently, the EVOLVE development on the propagation patterns focuses more on how to reuse the existing design and specification under the schema modifications and requirement changes, and how the existing propagation patterns can be reused or extended incrementally to cover the new requirements, especially when both structural and operational changes are required.

In short, the most interesting features of the EVOLVE development include:

- As an add-on CASE tool, it provides services for adaptive design and rapid prototyping of requirement specifications and for building or managing a C++ repository in an adaptive manner.
- As a design methodology, it exhibits promising features of adaptive specification techniques for object-oriented software evolution. On the side of structural specification, it provides a selection of style rules as basic means for construction of adaptive object-oriented schema design. On the side of behavioral specification, it promotes a novel conceptual programming technique – Propagation Patterns, and extends it by introducing the behavioral refinement mechanism for propagation patterns. These mechanisms are useful means for construction of adaptive schema design and for support of the operational level of reuse.
- The EVOLVE development has a formal basis. Both adaptive schema style rules and the reuse mechanisms for operational specifications by means of propagation patterns are formally developed with well-defined semantics. This formal basis provides a sound framework for the EVOLVE implementation and for further development of the ideas presented in this paper.

Acknowledgement

The author would like to thank Karl Lieberherr, Walter Hürsch for the discussions on the Demeter System/C++ and on the semantics of propagation patterns. My thanks also due to Roberto Zicari and Markku Sakkinen for the remarks on earlier versions of this paper.

References

[BCG+87] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, and Nat Ballou. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems*, 5(1):3 – 26, January, 1987.

[BD91] E. Bersoff and A. Davis. Impacts of life cycle models on software configuration management. *Communications of the ACM*, 34(8), 1991.

[Ber91] Paul Bergstein. Object-preserving class transformations. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 299–313, Phoenix, Arizona, 1991. ACM Press. SIGPLAN Notices, Vol. 26, 11 (November).

[BH93] Paul L. Bergstein and Walter L. Hürsch. Maintaining Behavioral Consistency during Schema Evolution. pages 176–193, Kanazawa, Japan, November 1993. JSSST. Vol. 742.

[BKKK87] J. Banerjee, W. Kim, H.J. Kim, and H.F. Korth. Semantics and implementation of schema evolution in object-oriented data bases, San Francisco, California. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, May 1987.

[Car84] Luca Cardelli. A semantics of multiple inheritance. In Gilles Kahn, David MacQueen, and Gordon Plotkin, editors, *Semantics of Data Types*, pages 51–67. Springer Verlag, 1984.

[CW85] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471, December 1985.

[HHG90] Richard Helm, Ian M. Holland, and Dipayan Gangopadhyay. Contracts: Specifying behavioral compositions in object-oriented systems. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 169–180, Ottawa, 1990. ACM Press. Joint conference ECOOP/OOPSLA.

[HLM93] Walter L. Hürsch, Karl J. Lieberherr, and Sougata Mukherjea. Object-oriented schema extension and abstraction. In *ACM Computer Science Conference, Symposium on Applied Computing*, pages 54–62, Indianapolis, Indiana, February 1993. ACM Press.

[LH90] Barbara Staudt Lerner and A. Nico Habermann. Beyond schema evolution to database reorganization. *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, 25(10):67–76, October 1990.

[Lie94] Karl J. Lieberherr. *The Art of Growing Adaptive object-oriented Software.* 1994. PWS Publishing Company, a Division of Wadsworth, Inc.

[LM92] Ling Liu and Robert Meersman. Activity model: a declarative approach for capturing communication behavior in object-oriented databases. In *Proceeding of the 18th International Conference on Very Large Databases*, Vancouver, Canada, 1992. Morgan Kauffman.

[LR88] Karl J. Lieberherr and Arthur J. Riel. Demeter: A CASE study of software growth through parameterized classes. *Journal of Object-Oriented Programming*, 1(3):8–22, August, September 1988. A shorter version of this paper was presented at the *10th International Conference on Software Engineering, Singapore, April 1988, IEEE Press*, pages 254-264.

[LX93] Karl J. Lieberherr and Cun Xiao. Formal Foundations for Object-Oriented Data Modeling. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):462–478, June 1993.

[LXSL91] Karl Lieberherr, Cun Xiao, and Ignacio Silva-Lepe. Propagation patterns: Graph-based specifications of cooperative behavior. Technical Report NU-CCS-91-14, Northeastern University, September 1991.

[SL90] A. Sheth and J.A. Larson. Federated database systems for managing distributed, heterogeneous, and autonomous databases. *ACM Transactions on Database Systems*, Vol. 22, No.3 1990.

[SZ87] Andrea Skarra and Stanley Zdonik. Type evolution in an object-oriented data base. In Bruce Shriver and Peter Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 393–416. The MIT Press, 1987.

[Zic91] R. Zicari. A framework of schema updates in an object-oriented database. In *Proceedings of International Conference on Data Engineering*, Japan, 1991. IEEE Press.