Facilitating Schema Evolution With Automatic Program Transformations

by
Michael M Werner

B.S. in Mathematics, Brooklyn College
M.S. in Mathematics, University of Illinois
M.S. in Computer Information Systems, Boston University

Submitted to the Faculty of the Graduate School
of the College of Computer Science
of Northeastern University
in Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

July 20, 1999

Dissertation Committee

Professor Kenneth Baclawski (Advisor)	.....
Professor Karl Lieberherr	.....
Professor William Clinger	.....
Doctor Ernesto Guerrieri (Inso Corporation)	.....

Dedicated to my wife Linda
and my children Jonathan and Daniel
for their encouragment and support.

Acknowledgements

I thank my advisor, Kenneth Baclawski for his support over the years that it took to complete this research. Ken was always quick to understand my own efforts, and came up with a great many useful suggestions, which I was able to incorporate. I would like to also thank Karl Lieberherr, who encouraged me from the outset. It was in Karl's classes, and also in the Demeter Seminar, which he directs, that I came up with much of the inspiration for my work. In that vein, I would also like to thank the other students and faculty members who participated in the seminar.

I would like to thank Ernesto Guerrieri, from whom I gained a great deal of insight into Java at a tutorial that he gave. Ernesto provided valuable input, particularly towards the end. I was able to learn a lot from his experiences as a practitioner in industry. Will Clinger helped me greatly in informal conversations, and in his painstaking examination of my work.

I learned a great deal from my Professors at the College of Computer Science. In particular, I would like to thank Agnes Chan, Larry Finkelstein, Ron Williams and Raoul Smith. While I did not take courses with them, I did hold many helpful informal conversations with Professors Betty Salzberg, Boaz Patt-Shamir, David Lorenz, Gene Cooperman, Robert Futrelle and Mitch Wand.

During the years of my graduate studies I was greatly helped by my employer, Wentworth Institute of Technology. Besides paying my tuition I was also given release time and travel funds, most importantly I was encouraged to keep striving, even though the goal often seemed far off.

Finally, I would like to thank my wife Linda and children Jonathan and Daniel, who stood by me during the long years of study and research, and occasionally gave up time on the family computer.

Abstract

Experience shows that even after programs have been designed, built and tested, change is more the norm than the exception. Consider a shared object-oriented persistent system built to serve the business needs of a company. Changes such as additions of classes or class members, renamings, retypings, etc. may be needed to introduce new applications, enhance existing ones, or integrate separately built systems. Additional changes such as factoring out duplicate code, rearranging the class hierarchy and delegating responsibilities to other classes, may be made for efficiency or clarity of design. The primary focus of this research is to provide a mechanism for making these changes easily and safely. Certain changes must be rejected, since they would introduce subtle errors, which might undermine the compilability, or the behavior of the system. The theoretical interest of this part of the research is to determine, for each kind of transformation, the necessary preconditions, which are required to preserve safety. Adherence to weak preconditions preserves type soundness, satisfying strong preconditions additionally guarantees behavior preservation. It is assumed that the source programs are written in Java, hence the preconditions are consequences of the language rules of Java [45].

A prototype tool, STP (Schema Transformation Processor) demonstrates the feasibility of this approach. With STP, a designer can reverse engineer a set of Java source programs to recover a design that is shown visually as a graph whose nodes represent classes, and whose arcs represent IS-A and HAS-A links. A language called Change Specification Language (CSL) is introduced for describing schema changes. CSL contains both low-level primitives for simple changes such as renaming a class and high-level primitives for broader changes such as factoring out common properties. High-level primitives are expanded into sequences of low-level ones at run-time. Program transformation is accomplished by parsing the source programs into an abstract syntax tree, then visiting the tree with transformation visitor objects, which update the code.

One type of transformation that often is needed, is to retrofit an existing program to support new applications. A running application is envisaged in terms of its navigation along certain HAS-A and IS-A links. The term itinerary, taken from the travel industry, is used as a metaphor for describing this navigation. STP has a mode, which makes it easy to specify an itinerary by clicking on the required links. The itinerary is then automatically attached to the source programs, by augmenting the classes involved with takeoff and land methods. These methods take a Visitor object as parameter. In accordance with the Visitor Design Pattern [44], the code to support navigation is cleanly separated from code which does work at the visited classes. A skeleton Visitor class is generated for each itinerary. The code to do a specialized task is encapsulated in one of its concrete subclasses. By creating additional subclasses, new functionality can easily be added on to existing itineraries, without disturbing the underlying classes.

A set of itineraries constitutes a subgraph of the graph representing the complete system. It is analagous to a view in a relational database. It is envisaged that by granting privileges on itineraries, access to a shared system can be limited on a need to know basis. By serving as targets for granting privileges, itineraries can play a role in securing shared object systems.

Contents

List of Figures

Chapter 1

Introduction

1.1 The Need For This Research

Studies [104] have shown that software maintenance constitutes about two-thirds of total software costs. Much of this maintenance can perhaps more correctly be called software evolution, as existing systems are adapted to take on new responsibilities and/or operate in a changed environment. For an object-oriented system, evolution often requires changes to the underlying object structure of the system in terms of classes, inheritance relationships, fields and operations. The database term for the underlying object structure is *schema*. Experience shows [106] that in practice there is often the need to change the original schema, either to correct design errors, to accommodate new applications, or to keep current with organizational changes. This is particularly true of shared persistent schemas which support a multitude of applications and users. Unfortunately changing the schema is often disruptive to existing programs, particularly ones which navigate through the system following paths of references between objects.

Even a change as simple as renaming a field is difficult in a large system. Besides changing the declaration of the field, all uses of the field inside method bodies must be located and changed as well. Since the same name may be used for different purposes in various parts of the program, a simple substitution in a text editor will not do the job. Furthermore, the change may be unsafe, in that it may result in a system which can no longer be compiled, or even if compilable, may differ in its behavior from the older system in unanticipated ways. This last may be the case, for example, if after renaming, the field now hides an inherited field with the same name.

This research investigates two kinds of program transformations. One kind is a schema transformation, such as adding an inheritance relationship, renaming a field or dropping a class. A range of schema transformations is identified, and for each one, an investigation is made into the conditions which must be satisfied before applying it so as to ensure safety. These conditions are consequences of both the nature of the transformation involved, and the rules of the programming language. Each transformation to a construct such as a field or operation signature also requires compensating changes to all the usages of the construct in method bodies. The necessary compensations are indicated for each type of transformation.

The ideas behind automatic schema transformations are introduced below in Section 1.2.

Another kind of program transformation, called an *itinerary*, adds new behavior or functionality to an existing system, involving modifications to a number of different classes. Itineraries, introduced in Section 1.3, allow for identification of paths through IS-A and HAS-A associations between classes, and provide automatic attachment of code to support navigation along the paths. An itinerary is actually a subschema, as such it is similar to a *view* in a relational database. The theoretical interest of itineraries is in their usefulness in extending view ideas into object-oriented systems. This is an active area of research, itineraries are compared to related work in Section 5.9.

1.2 STP (*Schema Transformation Processor*)

It is the contention of this thesis that automatic program transformation to reflect schema change is not only valid and feasible, but also forms the basis for a powerful and easy to use tool. Ease of use requires that the schema updates be describable in a simple, yet comprehensive language. Accordingly, a *Change Specification Language* (CSL) is introduced with descriptive commands such as RENAME, RETARGET, ADD, DROP, etc. These commands must reference schema constructs, accordingly a simplified object model, the *Implementation Object Model* (IOM) is introduced in Chapter 2. The IOM is a sparse model compared to the object model of UML [21], yet it does contain the essential schema constructs including all of those referenced by the CSL. The IOM also has a graphical representation, *IOM Graphs*, which completely represents the model using nodes, arcs and labels. An example of an IOM Graph is shown in Figure 1.1.

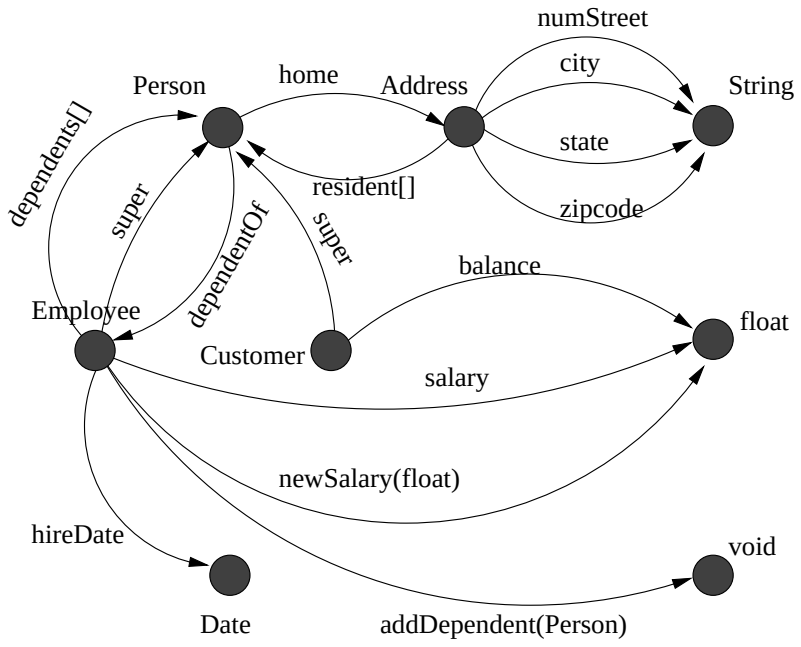

Figure 1.1: Sample IOM Graph

Unlike UML, the IOM Graph does not represent classes as boxes divided into compartments

for name, attributes, and operations. Rather, it uses labelled circles (nodes) for classes (as well as other types), and labelled arcs for attributes, operations, and for representing associations as HAS-A links. This is similar to the *arrow thinking* characteristic of category theory [12, 89], and allows visualization of compound programming language expressions such as `x.getY().z[5]` as paths in a graph.

A user may use the CSL to request a schema change which would be harmful in the sense that some programs would no longer compile, or perhaps produce altered behavior. Accordingly each CSL command has associated with it two preconditions:

1. **Weak precondition.** Satisfying it guarantees that the system will remain type safe in the sense that programs will continue to compile.
2. **Strong precondition.** Satisfying it additionally guarantees that the behavior of all programs will remain the same in terms of outputs and side effects.

In Chapter 3 these conditions and the rationale for them are painstakingly described. Usually, the preconditions can be stated visually and succinctly by referring to patterns of paths through the IOM Graph.

The CSL also includes several *Higher Level Primitives* (HLP's) such as FACTOR, FLATTEN, MERGE, and SPLIT which act as macros, expanding out into a number of *Lower Level Primitives* (LLP's) when used. These HLP's have their own preconditions.

As mentioned earlier, a prototype of the system, STP for *Schema Transformation Processor* has been built to transform Java programs. STP starts off by reverse engineering a list of Java source files so as to recover design information from the program declarations. This design (schema) is stored in a data structure which actualizes the IOM. This *working* IOM is later used to check preconditions. The Java programs themselves are parsed into an *Abstract Syntax Tree* (AST). Transformations are applied to both the IOM and the AST in tandem, so as to keep them consistent with each other. When all transformations have been applied, the programs are regenerated from the AST. This process is more fully described in Chapter 4.

1.3 Transforming Software Using Itineraries

The class model of a shared object system can be shown as a directed graph of nodes and arcs, the nodes are objects and the arcs represent generalization relationships, and associations between classes. A running application can be visualized in terms of its navigation along the arcs. In fact, it is feasible to add on new functionality to an existing system by modifying existing classes to provide the required navigational support; and then building visitor objects to do the specialized work required at the objects passed through. This is in accordance with the *Visitor Design Pattern* [44].

Retrofitting existing classes to support the navigational needs of new behaviors is an important transformation activity in support of software evolution. It requires first identifying the needed paths of navigation, and second, attaching new operations to the classes to support

the navigation. To facilitate the program designer in visualizing the navigation, an airline travel metaphor, namely the *itinerary* is used. The metaphor supports the idea of tours broken up into legs, with stops at airports. The legs themselves can be thought of as having takeoff and landing procedures, which might even include filters to limit access to selected instances of a class.

Itineraries are described fully in Chapter 5. One historical precedent for itineraries is adaptive programming, (Section 1.4.2), which similarly separates navigational concerns from behavioral ones. Another precedent is the view mechanism 5.9.4, commonly employed in databases. Like a view, an itinerary describes a subsystem of the whole, which is suitable to support a needed application. The set of itineraries required by a particular class of end-users can be combined to form a view, which roughly matches that class of users *need to know*.

It is also possible to specify a *virtual* itinerary, and later map it to a concrete one, replacing simple arcs in the virtual itinerary with paths of arcs in the concrete one. (See Section 5.7). Finally, it should be noted that much of the same change specification language (CSL) used by STP, can also be applied to itineraries, automating their update so as to conform to an updated object system.

1.4 Building on Prior Approaches

Prior approaches to the problem of evolving software divide up into three broad categories:

1. Prevention;
2. Facilitation;
3. Source Generation.

Prevention means eliminating the need to revise programs by somehow insulating them from schema change. Facilitation means providing automatic tools to revise programs. Source Generation means that the programmer works at a higher level than object-oriented programming languages, and a tool is provided to generate the source code when needed. This is a kind of hybrid between prevention and facilitation in that the forms which the developer works in are somewhat shielded from schema change, rerunning the tool with the new schema effectively transforms the source programs produced. Source Generation is described below in Section 1.4.2.

The prevention approach usually allows existing programs to continue to run as before by providing them with an older view of the shared system, while newer programs run against a revised view. Views which allow programs a limited filtered access to the global schema are a staple of relational databases, and there has been research into bringing them into object-oriented systems as well. This research is summarized in Section 5.9.4. The central idea of the view approach is to avoid the need for revising programs in the event of schema change by requiring application programs to access the schema through a view, and keeping the view constant when the underlying schema changes. The view is defined as a mapping

from a portion of the underlying schema, if that portion changes, the mapping is altered so as to compensate.

The view approach is predicated on the idea that each individual program in a shared system needs to see only a portion of the schema. The present research carries this idea further by introducing the notion of *itineraries*. The two premises of itineraries are that program behavior in terms of outputs and functionality can be cleanly separated from navigational concerns, and that it is possible to easily identify a subschema which is adequate for either a particular program, or even all the programs serving a particular class of end users. Itineraries are introduced later in this chapter, Section 1.3, and are described fully in Chapter 5.

The *facilitate* approach usually involves automatic program transformation. What is needed is a *smart* tool, which understands the structure of the system in terms of classes, class members, inheritance relationships, and variable scope. Schema changes such as renamings, adding or dropping class members, altering inheritance relationships, combining classes, or factoring out common members into a superclass, may introduce subtle problems which would disrupt existing applications. The *smart* tool needs to be able to detect the potential for such problems, and either compensate for them, or balk at making the changes.

Previous research on program transformation is summarized below in Section 1.4.1. The present research builds on this body of work, and relates it specifically to the problem of schema evolution. In order to do this it has been necessary to define a language for expressing schema change, the *Change Specification Language* (CSL), and an object model, the *Implementation Object Model* (IOM). These are introduced in Section 1.2, and described at length in Chapter 2. The IOM also has a graphical representation called an *IOM Graph*. There are many issues surrounding the preservation of program soundness and behavior. They are discussed in Chapter 3.

A prototype tool named STP (*Schema Transformation Processor*) has been built. STP works with programs written in Java, and most of the discussion as to the soundness of transformations also assumes that Java is used as the programming language. However there is no fundamental difficulty in extending both the tool and the theory to apply to other object-oriented languages such as C++ as well.

1.4.1 Facilitation - Research on Program Transformation

The importance of program transformation is underscored by the fact that it was one of the five areas covered by the working group on programming languages which formed part of the ACM Strategic Directions in Computing Research workshop held in Boston in June 1996 [48]. At that workshop Charles Consel [32] acknowledged that programs could be made *adaptable* by structuring them in terms of layers and modules, however went on to say that this often lowers performance. Program transformation can be used to produce efficient adaptations, perhaps using techniques such as partial evaluation and run-time code generation. Pettorossi and Proietti [88] stressed the importance of transformation rules and strategies and the introduction of basic transformation operations.

The Gen Voca Work on Program Transformation

A research group headed by Don Batory at the University of Texas has long been active in the field of automatic program transformation. One project, Rosetta [109] is a generator for programs which translate between related families of data languages, such as from SQL to Quel. The Unicron project [107] proposes a number of automatic transformations to be applied to source code so as to retrofit an existing program to utilize one or more of the design patterns popularized in [44]. Several transformations are described, including Inherit, Factory Method and Substitute. Transformations are specified using a template consisting of a description, arguments, initial state, target state and preconditions. The semantics of the desired transformation are specified by referring to class diagrams showing the initial and target states. Provided the preconditions are satisfied, transformations are then applied to all parts of the application program which are affected by the change.

1.4.2 Research on Source Generation

The underlying idea behind source generation schemes is to code at a higher level than programming languages, and thus avoid some of the *accidental difficulties* that occur at the programming language level. The merit of *raising the level of abstraction* was well illustrated for mathematics by Polya [90] back in 1949. The term *accidental difficulties* was introduced by Brooks [25] in 1987 to describe the type of problems in software engineering that can be overcome by using more powerful tools.

At Microsoft, a group led by Charles Simonyi is working on *Intentional Programming* (IP) [105]. IP programs are written as trees where each node represents, in a syntax free manner, a single intent of the programmer. So-called *reduction enzymes* attached to each node can be used to generate language-specific code when needed. Legacy source code can be parsed using legacy parsers into IP programs, and then transformed. In this way IP can be readily used for program transformation.

Subject and Aspect Oriented Programming

Subject-Oriented Programming (SOP) [47, 60, 82] is a technology which supports separate description of parts of C++ programs with automatic composition by the Watson Subject Compiler to produce a running program. SOP can be used to support among other things, the merging and splitting of design classes, a topic which is taken up here later on. SOP is in many respects similar to Aspect-Oriented Programming (AOP) [61], a technology being developed at Xerox/Parc which supports separate description of different aspects of a program such as the *persistence* aspect and the *distribution* aspect, with composition into a working program using an *Aspect Weaver*. Of these two technologies, SOP with its roots in the database world and its emphasis on seeing a shared system from different viewpoints, seems closest to the research proposed here.

Adaptive Programming

Adaptive programming[69, 71, 87] is another example of a source generation system. It stresses separation of the concerns of navigation from those of structure and behavior. This insulates the developer from many of the problems caused by schema evolution.

In traditional structured design [37], a process is represented by a single symbol in a data flow diagram, and in structured implementation often by a single function. Adaptive programming recognizes that in object-oriented systems a single behavior may require the cooperation of multiple objects and hence enters into a number of class definitions. This cooperation is the basis for the methodology known as CRC (Classes, Responsibilities, Collaborations) [113, 13], and of Holland's work on *contracts* [52, 50].

The challenge is to describe the required behavior in a single place and then to automatically distribute the necessary functionality into all the classes whose cooperation is needed to implement it. Adaptive programming meets this challenge by (1) specifying a traversal through the collaborating classes, and (2) specifying code to be executed when an object in one of these classes is entered or exited. The traversal is accomplished by following references and may require passing through so-called stepping stone classes at which no work needs to be done. The needed traversal code for these classes is automatically generated from a succinct traversal directive, by using path completion algorithms applied to the current class graph. The code specifying work to be done at cooperating classes is encapsulated in a visitor object, in accordance with the *Visitor Design Pattern* [44]. The visitor object contains before and after methods which can be invoked by the object being traversed. For an abstract Visitor class these methods can be empty, but concrete subclasses of Visitor use these methods to accomplish their work.

In Adaptive Programming it is assumed that a class graph exists. This is a directed graph whose nodes are classes, and whose edges represent either inheritance or aggregation relationships between the classes, however the details of the class graph are unknown at first.

As an example, here is a description of a traversal named listStudents which can be used to allow a professor to list the students in classes she is teaching.

```
Professor {
    traversal listStudents(Visitor v) {
        bypassing ->Professor,advisees,*
        to Student;
    }
}
```

Note that the traversal starts in the Professor class and ends in the Student class. There may be numerous stepping stone classes in between such as Course and Section, but since no important processing is needed at these classes they aren't mentioned in the traversal description. The traversal generates all paths from Professor to Student objects except for those explicitly bypassed, in this case paths that reach Student objects by following the edge labelled advisor from Professor objects.

The actual listing of students is done by a concrete visitor:

```
ListStudentsVisitor extends Visitor {
    before (Student s){
        System.out.println(s.name);
    }
}
```

Simple traversals can be joined end-to-end, (joining) or additively combined (unioning) to form complex ones. Recently, *strategies* have been introduced to represent abbreviated schemas useful for describing traversals. The mapping of a strategy to an actual class graph allows traversals defined on the strategy to be mapped to traversals on the class graph.

When an executable program is required a traversal is combined with the current version of the class graph in a process called *customizing*. Figure 1.2 shows two possible class graphs that could be used to customize the listStudents traversal.

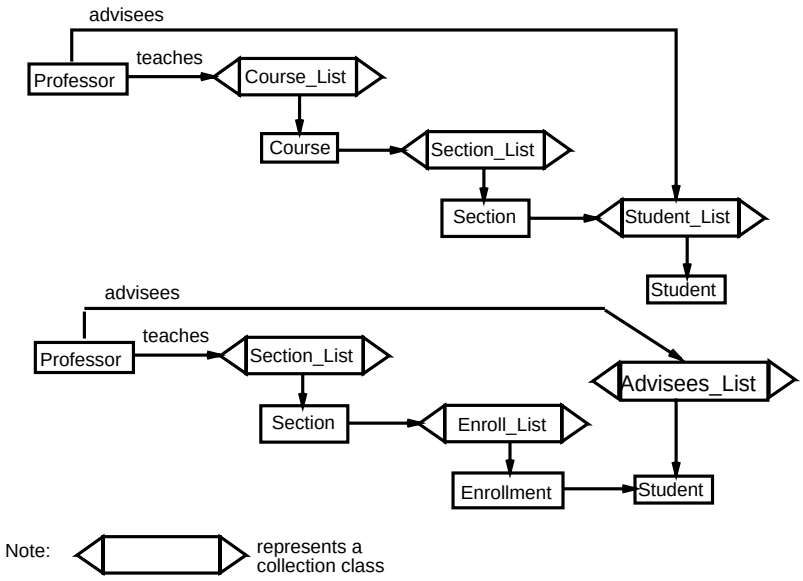

Figure 1.2: Two class graphs which could be used to customize the listStudents traversal

Demeter tools generate Java code from a list of traversals and a class graph. The adaptiveness stems from the fact that traversals are robust in the face of certain types of schema evolution, they are simply re-customized with the new class graph.

Palsberg [84] showed a polynomial-time algorithm for inferring from a traversal a set of constraints which any customizer must conform to, then building a representation of possible customizers, and finally, provided that set is non-empty, deriving a sample class graph(not necessarily minimal) which could serve as a customizer.

1.5 Why Java?

The examples in this dissertation, the discussion in Chapter 3 of the validity of transformations, the prototype programs demonstrating feasibility of both automatic schema updates and itineraries, all assume that Java is used as the programming language. While ideally it might be considered desirable to adapt a more general approach considering a range of object-oriented languages such as C++, Smalltalk, Eiffel, Java, CLOS, etc., in practice each of these languages has its own idiosyncrasies when it comes to issues such as objects vs. values, field types, operation signature matching, multiple vs. single inheritance, casting, and so on. The generalist is forever explaining modifications and exceptions that must be made to accommodate one language or the other. By concentrating on a single language the discussion will less often be sidetracked. In fact, this has ultimately been the pattern of previous investigators. Opdyke [81] is general for a while, but shortly settles on C++. Interestingly, most of his primitive "refactorings" were later implemented in Smalltalk [95]. Zicari [115, 114] and his associates [23, 41] state their transformation primitives in terms of the O_2 database.

Given that the approach is to reference a single language, why choose Java as that language? One reason is its widespread use and availability. Another is that Sun Microsystems [45], has issued a defacto language standard [45]. The Java grammar is readily available, and is simpler than that of C++. Also, the following language features greatly simplify the tasks at hand:

1. Java uses a single inheritance model, at least so far as classes are concerned.
2. All field and method declarations occur inside a class declaration.
3. Java programs do not use pointers. There is no pointer arithmetic, direct referencing using offsets, or other pointer associated evasions of the type system.

Finally, Java byte code can be readily decompiled, recovering most of the design information, so that the modeling and transformations proposed here can be accomplished even if the original source code is not available. (This decompilation can be deliberately thwarted by obfuscating the code.)

It should be stressed however, that the theory and techniques presented here are fairly general, in particular the prototype tools can be modified to work with other object-oriented languages such as C++.

1.6 The Rest of this Dissertation

This chapter has described some of the problems which occur when an object system is jointly shared by a number of end-users and their applications. In particular there is the problem that when some shared object structures must be modified to accommodate new or enhanced applications, other applications using these structures might be damaged. It

is the claim of this thesis that the damage can often be repaired automatically by requiring that structures be modified using a change specification language, the application of which would trigger updates to source code of all programs affected.

Chapter 2 presents the change specification language (CSL), along with an object model (IOM) which the changes refer to. There is also a graphical representation of the object model (IOM Graph) which is innovative in the sense that the essence of the model is captured in the arcs, rather than having complex nodes.

Chapter 3 is a thorough examination of each of the commands in the CSL, in terms of the sufficiency and necessity of their preconditions. These conditions are most often stated using patterns of paths in the IOM Graph.

Chapter 4 describes the development of the prototype program STP, which transforms Java source code in accordance with CSL commands. This program involves reverse engineering the source code to build a working version of the IOM for use in checking preconditions.

Another thrust of this thesis is that programs which share an object system can to a large extent be isolated from each other by:

1. Separating out navigational concerns from behavioral ones;
2. Defining the navigational needs for each program on a subgraph of the schema for the system.

Chapter 5 outlines a language called *Itineraries* for describing this navigation. A prototype tool *attaches* an itinerary to an IOM Graph. I believe there is great need and potential for this approach, particularly for use in object databases and other shared persistent systems. Itineraries provide the potential for limiting access to data on a *need to know* basis. They are also useful in adding new applications to existing systems, and for creating new shared systems in the first place by a kind of *schema integration*.

Finally, Chapter 6 summarizes the contribution of this work, and points to some new directions for future research.

Chapter 2

The Implementation Object Model and Change Specification Language

This chapter introduces an object model, called the *Implementation Object Model*, and a language, the *Change Specification Language*, which uses the model for describing changes to an object system.

2.1 Introduction

Software engineers often use graphical models for visualizing object systems. Chen's *Entity Relationship Model* [31] for visualizing database schemas, was later expanded by Rumbaugh, et. al. into the *OMT Object Model* [97], and still later incorporated into the Unified Modeling Language [21]. Another widely used object model, ODMG-93 [29] has been developed by the industry consortium known as the Object Data Management Group. These models strive for generality, although both ODMG and UML have bindings to specific languages including C++, Java and Smalltalk.

Both the ODMG and UML object models have grown to be large, and contain a great deal more information than is needed for the transformations envisioned in this work. Furthermore, questions about the exact semantics of the models are still being worked out [4]. For these reasons, a simplified object model, the *Implementation Object Model* (IOM), is introduced here which is bound closely to the Java language.

The IOM has its own visual representation, known as an *IOM Graph*, introduced in Section 2.3. However, to provide continuity to software designers and engineers, a related graphic, known as the *Design View* is also introduced in Section 2.4. This is a simplified version of the OMT object model. The conversion from IOM Graph to Design View is described. The IOM allows the expression of constraints (invariants) that reflect the rules of the Java Language, and ensure the viability of the model in terms of its being successfully translatable into Java code.

The chief business of this research is to transform programs, which collaborate to build an

object-oriented system. To do this requires a language for describing the transformations. This chapter presents a *Change Specification Language* (CSL), which expresses program transformations by referencing the IOM.

The CSL has two types of operations, known as *Lower Level Primitives* (LLP's), and *Higher Level Primitives* (HLP's). An LLP describes a single alteration of the IOM. An HLP describes a broader type of alteration such as factoring out common elements of two classes into a new abstract class from which both will inherit. An HLP expands into a sequence of LLP's.

A grammar is presented for the CSL in Section 2.10. Section 2.5 gives an intuitive description of each of the LLP commands, including illustrative examples of each. Section 2.7 does the same for HLP commands. Chapter 3 gives a more formal specification of each command, including discussion of necessary, and in some cases, sufficient preconditions, and also the postconditions that will attain after the command is carried out. In Chapter 4 the mechanics of carrying out these transformations is described, including the checking of preconditions.

2.2 The Implementation Object Model

The Implementation Object Model (IOM) is a view of an object-oriented system as seen from a programming language, rather than a software engineering perspective. The model presented here is oriented towards Java, although it would also be useful with C++.

The IOM follows roughly the model in the O_2 database [115] in terms of constructs used. However, unlike the O_2 model, the IOM model is designed to be easily implementable as a data structure which can be queried to determine the state of the object system. This facility is used by the prototype program STP (described in Chapter 4) to check preconditions for transformations. Following transformation, the IOM model is itself updated to correctly model the new system.

The IOM captures the essential schema information concerning classes, their fields and operations, and inheritance relationships (IS-A links) between classes. Fields are classified either as attributes or HAS-A links. The attributes take on primitive values such as int and float, the HAS-A links take on values which are references to objects.

Attributes and HAS-A links are allowed to be multiple-valued. For Java 1.1, this reflects the fact that fields can be arrays. If, and when parameterized types such as $List < A >$, or $Set < A >$ are introduced into Java [77, 80], they too can be represented by multi-valued attributes or HAS-A links.

The graphical representation of the IOM shows classes as nodes and IS-A's, fields, and operations as arcs. See Section 2.3. Finally, the IOM can record a set of constraints (invariants).

First some preliminaries. An IOM model is built from the declarations of classes, fields and methods found in a set of Java source files that comprises a *program*. C_{id} represents the set of identifiers used as class names, including names of classes from explicitly imported

packages, as well as the implicitly imported *java.lang*. P_{id} represents the set of identifiers used to name attributes, HAS-A links and operations. *Prim* is the set of primitive types, which in Java are *boolean, char, byte, short, int, long, float* and *double*.

Finally, $Types = C_{id} \cup Prim$. For $t \in Types$, Primitive(t) is true if $t \in Prim$, false otherwise.

In the definitions that follow, set and list have their usual meanings.

2.2.1 The IOM Defined

An IOM is a tuple $(\mathcal{C}, \mathcal{H}, \mathcal{I}, \mathcal{R})$, where:

- $\mathcal{C}$ is a set of ClassDefs;
- $\mathcal{H}$ is a set of HAS-A's;
- $\mathcal{I}$ is a set of IS-A's;
- $\mathcal{R}$ is a set of Restrictions.

A ClassDef is a tuple (name, attributes, operations), where:

- name $\in C_{id}$;
- attributes is a set of Attribute;
- operations is a set of Operation.

An Attribute is a tuple (name, type, multiplicity) where:

- name $\in P_{id}$;
- type $\in Prim$;
- multiplicity = 1 if single-valued, 0 if multi-valued.

An Operation is a tuple (name, parameterList, returnType) where:

- name $\in P_{id}$;
- A parameterList is a list of Types, possibly empty;
- returnType $\in Types \cup$ "*void*".

A HAS-A is a tuple (sourceClass, targetClass, linkName, multiplicity, inverseLinkName [1]) where:

- sourceClass $\in C_{id}$;
- targetClass $\in C_{id}$;
- linkName $\in P_{id}$;

[1] inverseLinkName is optional, it indicates the name of an inverse HAS-A link which together with this link form a binary association between the classes named as Source and Target. While inverse links are not used in Java or C++ they are part of the proposed ODMG standard [29].

- multiplicity = 1 if single-valued, 0 if multi-valued;
- inverseLinkName $\in P_{id}$.

An IS-A is a tuple (subclass, superclass) where:

- subclass $\in C_{id}$;
- superclass $\in C_{id}$.

A Restriction is a predicate which can be evaluated by examining the current state of the IOM.

Finally, note that an *operation signature* consists of a name and parameter list only. Signatures are used for matching operations.

2.2.2 IOM Notations and Conventions

If $(A, B) \in \mathcal{I}$, we write A$\Longleftarrow$B, meaning A IS-A B. We write A$\stackrel{\star}{\Longleftarrow}$B if A = B, or there is a chain of 1 or more IS-A links, A$\Longleftarrow$C, C$\Longleftarrow$D, ... $\Longleftarrow$B.

If C $\in C_{id}$ then the attributes and operations defined in ClassDef(C) are referred to as:

- attributes(C)
- operations(C)

Additional attributes and operations may be inherited. We write:

- Attributes(C);
- Operations(C);

to include both locally defined and inherited attributes and operations at C. Similarly we write:

- hasas(C) = $\{h \in \mathcal{H} \mid sourceClass(h) = C\}$;
- Hasas(C) to also include inherited HAS-A links;

and

- fields(C) = $attributes(C) \cup hasas(C)$;
- Fields(C) = $Attributes(C) \cup Hasas(C)$.

We can also speak of the *origin* of a class member, such as an attribute, operation, or HAS-A, as being either the class itself, or the nearest superclass which defines the member. For an operation, defines is taken to mean supplying a method body, as well as the header. Let m be a member of class C. If m is defined in C, then origin(m) = C, else origin(m) = D if m is defined in D, and m is not defined in any class K, where:

$$C \stackrel{\star}{\Longleftarrow} K \stackrel{\star}{\Longleftarrow} D$$

2.2.3 Restrictions

$\mathcal{R}$ includes restrictions (invariants) that reflect the rules of the Java Language, and ensure the viability of the model in terms of its being successfully translatable into Java code. To some extent, they are patterned after a list found in Zicari [114].

- R1: *Unique class names:*

$$\forall C_1, C_2 \in \mathcal{C}, Name(C_1) = Name(C_2) \Longrightarrow C_1 = C_2$$

- R2: *Acyclicity:* $\mathcal{I}$ is acyclic.

$$\forall C_1, C_2 \in \mathcal{C}, C_1 \stackrel{\star}{\Longleftarrow} C_2 \wedge C_2 \stackrel{\star}{\Longleftarrow} C_1 \Longrightarrow C_1 = C_2$$

- R3: *Unique field names:*

$$\forall C \in \mathcal{C}, \forall f_1, f_2 \in fields(C), Name(f_1) = Name(f_2) \Longrightarrow f_1 = f_2$$

- R4: *Unique operation signatures:*

$$\forall C \in \mathcal{C}, \forall o_1, o_2 \in operations(C), signature(o_1) = signature(o_2) \Longrightarrow o_1 = o_2$$

- R5: *Single origin:* All fields and operations available at a class have a single origin.

$$a \in Fields(C) \Longrightarrow \mid origin(a) \mid = 1$$
$$o \in Operations(C) \Longrightarrow \mid origin(o) \mid = 1$$

- R6: *Operations do not override return type:*

$$o_1 \in operations(C), o_2 \in operations(D), C \stackrel{\star}{\Longleftarrow} D,$$
$$signature(o_1) = signature(o_2) \Longrightarrow returnType(o_1) = returnType(o2)$$

  This property is enforced in the Java language.

2.3 IOM Graphs

There is a graphical representation of the Implementation Object Model which is designed to aid in visualizing relationships between the classes which comprise the system. The graphical representation will also prove useful for interpreting the usage traces introduced in Chapter 3, and the itineraries of Chapter 5.

An IOM graph consists of nodes and directed arcs.

2.3.1 IOM Graph Nodes

A node is either:

- A reference node, *Ref* for short, standing for a class definition. It is labelled by the class name.
- A primitive node, *Prim* for short, standing for a primitive type such as int or boolean, and labelled by the type name.
- A void node, written *Void* and labelled "void", providing a target for operations which do not return a value.

All nodes are shown as small filled circles, labelled either "void", or by the class name or primitive type they represent.

2.3.2 IOM Graph Arcs

IOM Graph arcs are used to represent inheritance relationships between classes, as well as all members of a class including attributes, HAS-A's and operations. Arcs are directed, and usually are drawn as arcs of a circle. Arc labels are important, the label "super" indicates an inheritance arc, and operations are distinguished from fields by labelling them using parentheses. For the purpose of IOM graphs, operations are divided into 3 categories:

1. RefOp - An operation whose return type is a reference to a class.
2. PrimOp - An operation with primitive return type.
3. VoidOp - An operation which returns void.

IOM arcs are described in Figure 2.1.

Multiple-valued attributes and HAS-A links are additionally labelled by appending "[]" to their names. So, $M \xrightarrow{n} N$ is the notation for a single-valued arc from M to N, $M \xrightarrow{n[]} N$ denotes a multiple-valued arc. Note that each IOM node N has a unique identity arc,

Figure 2.1: Arcs in IOM Graphs

Arc Type	Labelling	Joining
IS-A	"super"	Ref $\longrightarrow$ Ref
Identity	(*nodeName*)	Node $\longrightarrow$ Node
HAS-A	name	Ref $\longrightarrow$ Ref
attribute	name	Ref $\longrightarrow$ Prim
RefOp	signature	Ref $\longrightarrow$ Ref
PrimOp	signature	Ref $\longrightarrow$ Prim
VoidOp	signature	Ref $\longrightarrow$ Void

$N \xrightarrow{(N)} N$, which maps objects of class N to themselves. The use of parentheses in the labelling of an identity arc suggest a *cast* from a class to itself.

The IOM arcs in Figure 2.1 are called *simple*. To identify an arc unambiguously requires knowing the node which is the source of the arc as well as its label. For that purpose, the scope resolution operator "::" is borrowed from C++ to form *arc handles* from the name of the source node, and the arc label. So, for example: A::getX() is the handle for the arc in Figure 2.2 emanating from A with label "getX()".

IOM graphs also have *composite* arcs, composed of two or more simple arcs joined end to end. Composite arcs are labelled by concatenating the labels of the simple arcs using "::" as a separator, for example: "C::super::x". Arc handles are composed as follows:

Given $X \xrightarrow{label1} Y$, and $Y \xrightarrow{label2} Z$, where X, Y, and Z are nodes, the composite of X::label1 and Y::label2 is X::label1::label2. (Note that :: is left associative).

Normally, to avoid clutter, neither composite arcs nor identity arcs are drawn in IOM Graphs diagrams.

Following the rules of Java, it is clear that arcs emanating from the same node are uniquely labelled. "super" is a reserved word, and cannot be used to name fields. There can be only one "super" for a Java class. The only arcs labelled with "(" as the first character are Identity arcs. HAS-A links and attributes are both fields, and in Java, fields declared in a class have unique names. The operation signatures used to label operations must likewise be unique, and are different from field names because they have parentheses. As a consequence of this, all arc handles are unique.

Figure 2.2 illustrates the IOM graph induced by the toy Java program in Figure 2.3.

The strong visual quality of an IOM Graph makes it an ideal target to be manipulated by a CASE tool. For example, although CSL commands are stated as sentences in a language, a CASE graphics editor could allow a program designer to manipulate the arrows by dragging, with the tool generating the required sentences.

IOM Graphs extend a family of graphical representations of database schemas. For example, at the University of Antwerp, GOOD [46] similarly uses nodes for classes, and arcs for showing fields and IS-A relationships. GOOD does not however, show operations as arcs as does the IOM Graph.

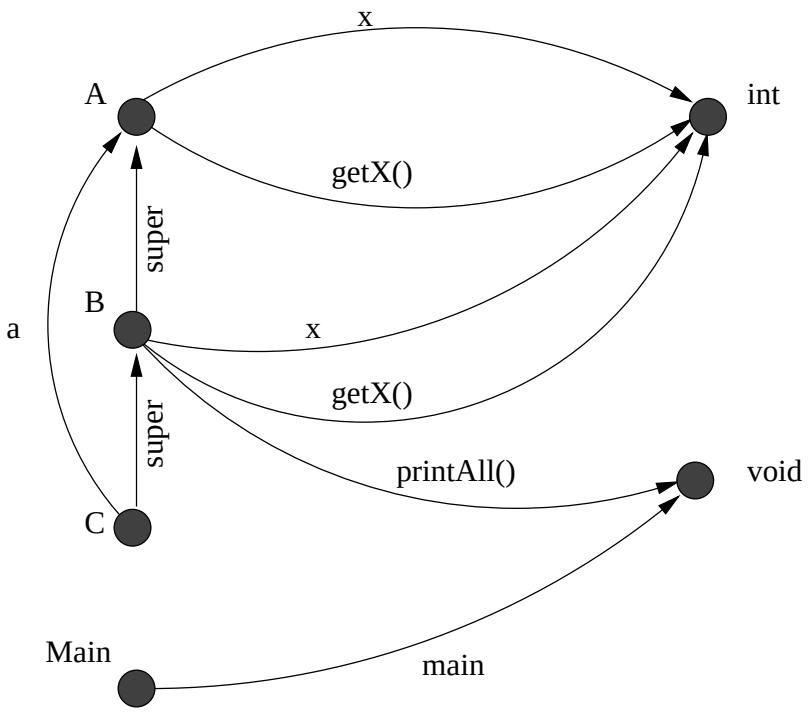

Figure 2.2: The IOM Graph Induced by the Java Program in Figure 2.3

IOM Graphs can also be used to characterize the clients and servers of a particular class. A *client* of class C is any class X for which there is at least one arc from X to C. These are the classes which derive some services from C. Note that this includes subclasses of C. Similarly, class Y is a *server* for C if there is at least one arc from C to Y.

2.3.3 IOM Graph Arcs Viewed as Functions

The IOM Graph arc M $\xrightarrow{n}$ N, where M $\in \mathcal{C}$, can be viewed as a function mapping objects of type M into either objects of type N, if N is a class, or values of type N, if N is a primitive type. The arc M $\xrightarrow{n[]}$ N represents a set of functions f[i]:M $\longrightarrow$ N, where i ranges over some index set I.

Consider, for example, the Java declaration:

```
class M{
    N[] n = new N[5];
}
```

The IOM Graph arc M $\xrightarrow{n[]}$ N represents a single multi-valued HAS-A, or viewed from a functional perspective, a set of 5 functions, n[i]:M $\longrightarrow$ N, i = 0,1,2,3,4.

If A,B and C are IOM nodes, with A and B classes, a an object in A, and the IOM arcs:

$$A \xrightarrow{b1} B$$

```
import java.io.*;
class A{
    protected int x = 10;
    int getX(){return x;}
}
class B extends A{
    int x = 20;
    public void printAll(){
        System.out.println("b.x = "+this.x);
        System.out.println("b.super.x = "+super.x);
    }
    int getX(){return x;};
}
class C extends B{
    C(){super(); a = new A();}
    A a;
}
class Main{
    public static void main(String argv[]){
        B b = new B();  b.printAll();
        System.out.println("((A) b).x = " + ((A) b).x);
        System.out.println("(new B()).x "+(new B()).x );
        A ab = new B();
        System.out.println("ab.x = "+ ab.x);
        System.out.println("ab.getX() = "+ ab.getX());
        C c = new C();
        System.out.println("((A) c).x = "+((A) c).x);
        System.out.println("c.a.getX() = " + c.a.getX());
    }
}
```

Figure 2.3: Main.java - A Toy Program

$$A \xrightarrow{b2[]} B \text{ indexed by I}$$

$$B \xrightarrow{c1} C$$

$$B \xrightarrow{c2[]} C \text{ indexed by J}$$

then the 4 composite arcs:

$$A \xrightarrow{b1::c1} C$$

$$A \xrightarrow{b1::c2[]} C \text{ indexed by J}$$

$$A \xrightarrow{b2[]::c1} C \text{ indexed by I}$$

$$A \xrightarrow{b2[]::c2[]} C \text{ indexed by I,J}$$

are formed, with the following meanings when viewed from a functional perspective:

1. b1::c1(a) = c1(b1(a));
2. b1::c2[] is a set of functions b1::c2[j]:A $\longrightarrow$ C indexed by J, where b1::c2[j](a) = c2[j](b1(a)) for $j \in J$;
3. b2[]::c1 is a set of functions b2[i]::c1:A $\longrightarrow$ C indexed by I, where b2[i]::c1(a) = c1(b2[i](a)) for $i \in I$;
4. b2[]::c2[] is a set of functions b2[i]::c2[j]:A $\longrightarrow$ C indexed by I and J, where b2[i]::c2[j](a) = c2[j](b2[i](a)) for $i \in I, j \in J$.

2.4 The Design View of an IOM

The IOM graphs in Section 2.3 are at variance with the object model graphing techniques used in popular software engineering methodologies such as OMT [97], and UML [34]. In some ways, the OMT/UML style is easier to read; there is much less arc clutter. For these reasons, there is an alternative graphing technique, known as *Design View*, for IOM's. Similar to OMT, the design view shows classes as rectangles divided into 3 compartments, for the class name, attributes, and operations respectively. Binary associations are shown as lines connecting rectangles, along with labels for association names. In addition, near each end of a binary association are labels indicating the role name and multiplicity of the class at that end. Inheritance of classes is shown using crows feet or small triangles pointing to the superclass. For simplicity more involved constructs such as ternary and higher associations are not represented, nor is there any attempt to distinguish aggregation associations from non-aggregation ones. Figure 2.4 illustrates a design view for an IOM which models a business.

The design view can show different levels of detail, as desired. An attribute can be written as name only, or as a name annotated with its type. Similarly, an operation can be shown just using its name, or using its signature and return type.

A *binary association* is drawn as a line between two class rectangles; the classes are said to be the *participants* in the association. The association may be labelled, although such labels

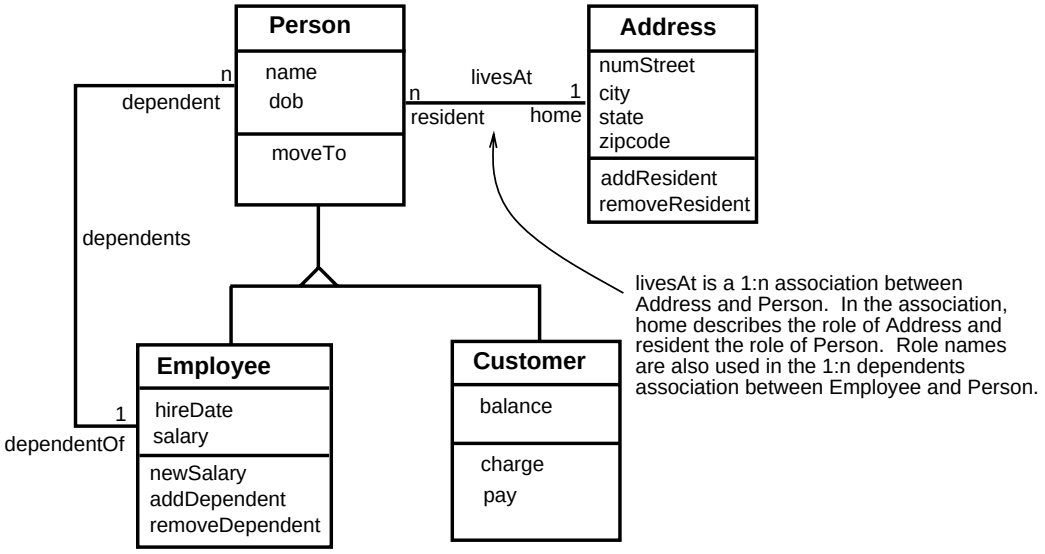

Figure 2.4: A design view of part of a business

have no counterpart in the IOM. Where the association line meets a participant, there may be additional annotations to indicate the role name and multiplicity of the participant. So, for example, in Figure 2.4, the association "livesAt" has Person and Address as participants, Person playing the role of resident, and Address the role of home. The multiplicities n and 1 indicate that an address may have many residents, but a Person only one home.

2.4.1 Transforming the Design View to the IOM Graph

1. Each design rectangle representing a class becomes a Ref node, labelled by its class name.
2. Additional nodes representing the primitive types and Void are drawn.
3. An inheritance association (crows feet), is resolved into one or more IS-A arcs pointing to the superclass, one from each subclass.
4. An attribute becomes an arc from the Ref node representing its ClassDef, to the node representing its type. It is labelled with its name.
5. An operation is shown as an arc from the ref node representing its ClassDef, to the node representing its return type. It is labelled with its signature.
6. A binary association is resolved into a pair of HAS-A links using the role name and multiplicity of each participant. So, for example, an association between classes A and B is shown as two arcs, one from A to B, labelled by the role name of B, and the other from B to A, labelled by the role name of A. This is illustrated in Figure 2.6.

Figure 2.5 shows an IOM matching the Design View shown in Figure 2.4. (Some operations have been omitted). Note that one to many references are indicated by arc labels with "[]"

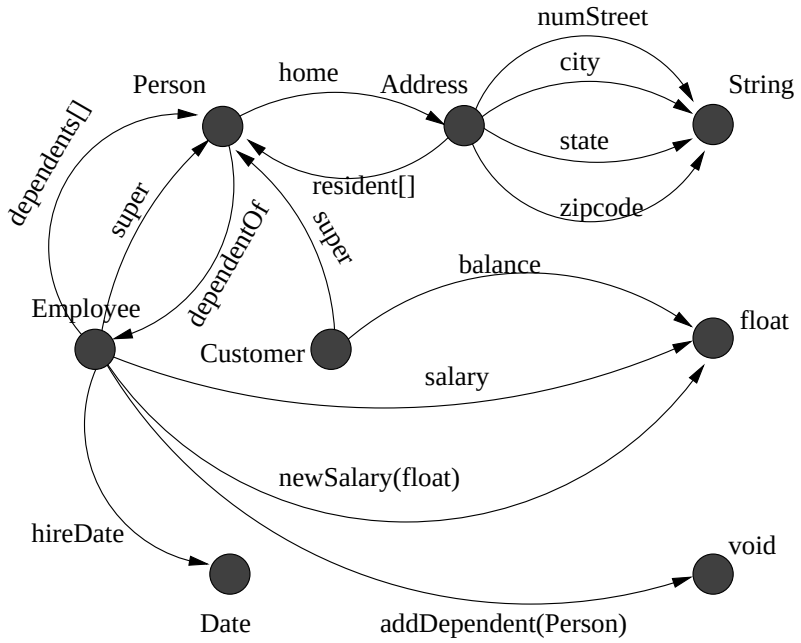

Figure 2.5: IOM Graph Matching Figure 2.4

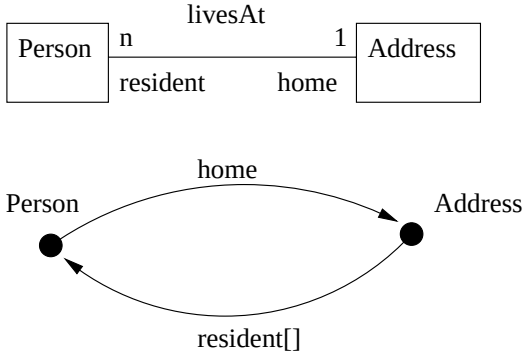

Figure 2.6: Converting a binary association to a pair of HAS-A links

suffixes. In Java these could be implemented at present using arrays or collection classes such as java.util.Vector and java.util.Hashtable. Should parameterized types become available in Java as proposed [77, 80], collections such as Set$< Person >$ could be used.

2.4.2 Transforming the IOM Graph to the Design View

The only sticky point in this transformation is the pairing up of HAS-A links to form associations. Recalling its definition, a HAS-A link has an optional field for inverseLinkName. Where this has been filled in for at least one HAS-A in the pair, they can be matched correctly. Otherwise, either human intervention is needed to do the matching, or two unrelated associations must be drawn. The HAS-A link names become role names for the participants. Recall that in the IOM Graph, multiplicity 1 indicates a single-valued link,

and 0 a multi-valued one. In the design view, these are shown as "1" and "n", respectively.

As for the other constructs:

1. Each IOM class is drawn as a rectangle.
2. IOM IS-A links with a common superclass are combined and drawn using the crows feet convention.
3. Attributes are placed in the attributes compartments, rather than being drawn as arcs.
4. Operations are placed in the operations compartments, rather than being drawn as arcs.

2.5 Lower Level Primitives (LLP's) Listed

A number of operations are designed as transformations of the Implementation Object Model. The operations can be expressed using a Change Specification Language (CSL) modeled after [79]. The CSL consists of *Lower Level Primitives* (LLP's) which roughly match up with operations directly supported by the object-oriented databases such as O_2 [10], and *Higher Level Primitives* (HLP's) which require expansion into LLP's before they can be executed. The basic operations ADD and DROP are named after similar DDL operations in SQL; additional operations including RENAME, RETARGET, DELEGATE and RECLAIM are also used.

Figure 2.7: Construct Update-Type Matrix

Construct	Add	Drop	Rename	Retarget	Remult	Delegate	Reclaim	Lift	Lower
Class	X	X	X						
Attribute	X	X	X	X	X	X	X	X	X
Operation	X	X	X	X		X	X	X	X
HAS-A	X	X	X	X	X	X	X	X	X
IS-A	X	X							

Figure 2.7 summarizes the lower-level primitives by the type of update. The target of an attibrute or HAS-A refers to its type; the target of an operation is its return type. Attributes and HAS-A's can be remultiplied, meaning changing from single-valued (multiplicity = 1) to multiple-valued (multiplicity = 0), or vice-versa.

The Lower Level Primitives are as follows:

1. Changes to classes
   (a) ADD CLASS *ClassName*
   (b) DROP CLASS *ClassName*
   (c) RENAME CLASS *OldClassName* NEW NAME *NewClassName*

2. Changes to attributes

   (a) ADD ATTRIBUTE *attributeName* SOURCE *ClassName* TARGET *attributeType*

   (b) DROP ATTRIBUTE *attributeName* SOURCE *ClassName*

   (c) RENAME ATTRIBUTE *oldAttributeName* SOURCE *ClassName* NEW NAME *new attributeName*

   (d) RETARGET ATTRIBUTE *attributeName* SOURCE *ClassName* NEW TARGET *new attributeType*

   (e) REMULTIPLY ATTRIBUTE *attributeName* SOURCE *ClassName* NEW MULTIPLICITY (0 | 1)

3. Changes to the HAS-A Hierarchy

   (a) ADD HAS-A *has-aName* SOURCE *SourceClassName* TARGET *TargetClassName* MULTIPLICITY (0 | 1)

   (b) DROP HAS-A *has-aName* SOURCE *SourceClassName*

   (c) RENAME HAS-A *oldHas-aName* SOURCE *SourceClassName* NEW NAME *newHas-a name*

   (d) REMULTIPLY HAS-A *has-aName* SOURCE *SourceClassName* NEW MULTIPLICITY (0 | 1)

   (e) RETARGET HAS-A *has-aName* SOURCE *SourceClassName* NEW TARGET *NewTargetClassName*

4. Changes to operations

   (a) ADD OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) TARGET *typeName*

   (b) DROP OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...);

   (c) RENAME OPERATION *oldOperationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ..) NEW NAME *newOperationName*

   (d) RETARGET OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) NEW TARGET *newType*

5. Changes to the class hierarchy

   (a) ADD IS-A *SubclassName SuperclassName*

   (b) DROP IS-A *SubclassName SuperclassName*

6. Moving a field [2] or operation

   (a) DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *has-aName*

[2] A field is either an attribute or a HAS-A

(b) DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*

(c) RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *has-aName*

(d) RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*

(e) DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*

(f) DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

(g) RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*

(h) RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

(i) LIFT FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*

(j) LIFT OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*

(k) LOWER FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*

(l) LOWER OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*

2.6 Lower Level Primitives Described

The LLP's are now individually described, accompanied by examples. However, a detailed discussion of the impacts of the transformations is deferred until Chapter 3.

2.6.1 ADD CLASS

- **Primitive:** ADD CLASS *ClassName*.
- **Example:** ADD CLASS X.
- **Description:** A new class is added to the system.

2.6.2 DROP CLASS

- **Primitive:** DROP CLASS *ClassName*.
- **Example:** DROP CLASS X.
- **Description:** An existng class is dropped from the system (provided these are no usages of it).

2.6.3 RENAME CLASS

- **Primitive:** RENAME CLASS *OldClassName* NEW NAME *NewClassName*
- **Example:** RENAME CLASS A NEW NAME B
- **Description:** A class is renamed. All usages of the class name in expressions, such as casts and constructor invocations are also renamed.

2.6.4 ADD ATTRIBUTE

- **Primitive:** ADD ATTRIBUTE *attributeName* SOURCE *ClassName* TARGET *attributeType*
- **Example:** ADD ATTRIBUTE x SOURCE B TARGET double
- **Description:** An attribute (field with primitive type) is added to the class declaration for *ClassName*.
- **Note:** Adding an attribute can hide an inherited field. Expressions which formerly accessed that field must be adjusted to include a cast. This is shown in Figure 2.8.

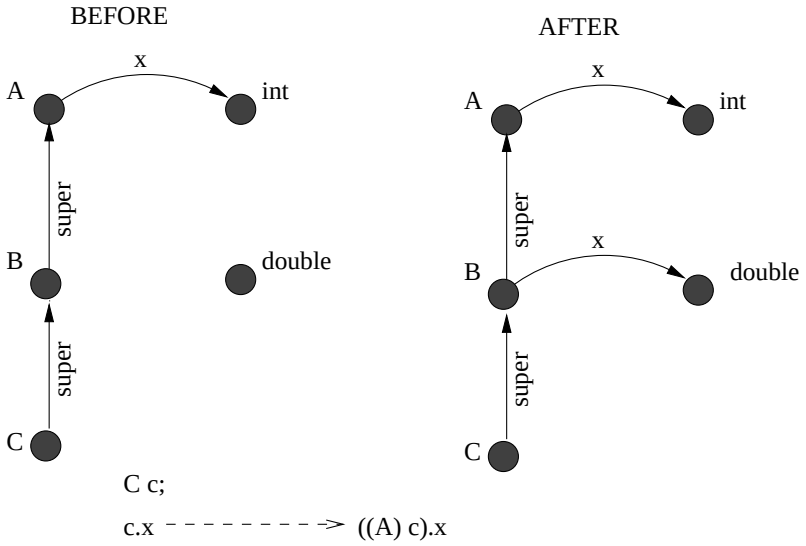

Figure 2.8: Before and after adding attribute x to class B. The primary expression, c.x, where c is an object of type C, is adjusted to include a cast.

2.6.5 DROP ATTRIBUTE

- **Primitive:** DROP ATTRIBUTE *attributeName* SOURCE *ClassName*
- **Example:** DROP ATTRIBUTE x SOURCE B
- **Description:** The attribute named *attributeName* is dropped from the class named *ClassName*.

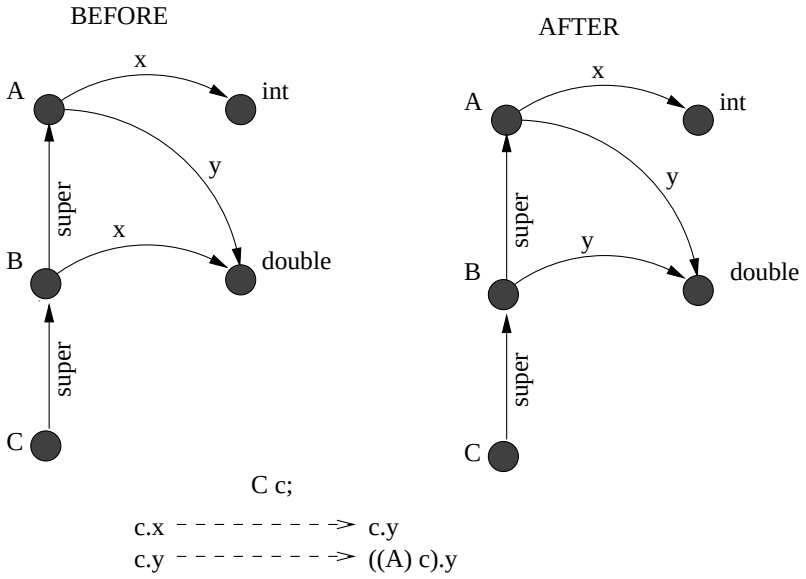

Figure 2.9: Before and after renaming attribute x in class B to y

2.6.6 RENAME ATTRIBUTE

- **Primitive:** RENAME ATTRIBUTE *oldAttributeName* SOURCE *ClassName* NEW NAME *new attributeName*
- **Example:** RENAME ATTRIBUTE x SOURCE B NEW NAME y
- **Description:** The field representing the attribute is renamed in the class declaration for *ClassName*.
- **Note:** As shown in Figure 2.9, renaming a field might hide an inherited field, or expose a previously hidden one. This requires compensating changes in all expressions which make use of the attribute. Some of the changes are shown at the bottom of the figure. Chapter 4 details the changes that need to be made.

2.6.7 RETARGET ATTRIBUTE

- **Primitive:** RETARGET ATTRIBUTE *attributeName* SOURCE *ClassName* NEW TARGET *newType*
- **Example:** RETARGET ATTRIBUTE a SOURCE X NEW TARGET boolean
- **Description:** The field representing the attribute is given a new type. See Figure 2.10.

2.6.8 REMULTIPLY ATTRIBUTE

- **Primitive:** REMULTIPLY ATTRIBUTE *attributeName* SOURCE *ClassName* NEW MULTIPLICITY (0 | 1)

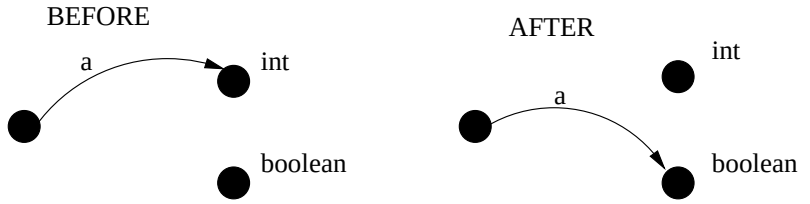

Figure 2.10: Before and after retargeting attribute a in class X from int to boolean

- **Example:** REMULTIPLY ATTRIBUTE a SOURCE X NEW MULTIPLICITY 0
- **Description:** The field representing the attribute is assigned a new multiplicity, either multi or single-valued.

2.6.9 ADD HAS-A

- **Primitive:** ADD HAS-A *has-aName* SOURCE *SourceClassName* TARGET *Target-ClassName* MULTIPLICITY (0 | 1) [INVERSE *inverseHas-aName*]
- **Example:**
  1. ADD HAS-A h1 SOURCE X TARGET Z MULTIPLICITY 0
  2. ADD HAS-A h1 SOURCE X TARGET Z MULTIPLICITY 0 INVERSE h2 (Using the optional inverse. This is illustrated in Figure 2.11.)
- **Description:** A new HAS-A field named h1 is drawn from X to Z. If the optional inverse clause is included, an inverse HAS-A with multiplicity 1 is drawn from Z to X. The multiplicity of the inverse can be changed later, using *REMULTIPLY HAS-A*.

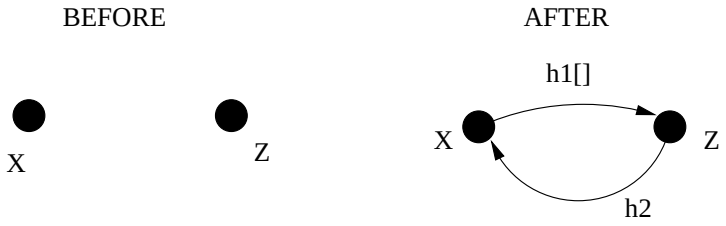

Figure 2.11: Before and after adding a HAS-A link with inverse

2.6.10 DROP HAS-A

- **Primitive:** DROP HAS-A *has-aName* SOURCE *ClassName*
- **Example:** DROP HAS-A x SOURCE B
- **Description:** The has-a named *has-aName* is dropped from the class named *Class-Name*.

2.6.11 RENAME HAS-A

- **Primitive:** RENAME HAS-A *oldHas-aName* SOURCE *SourceClassName* NEW NAME *newHas-aName*
- **Example:** RENAME HAS-A x SOURCE B NEW NAME y
- **Description:** The name of the HAS-A is changed to y. If the HAS-A has an inverse link, the inverse's inverse link is also changed to y. See Figure 2.12.

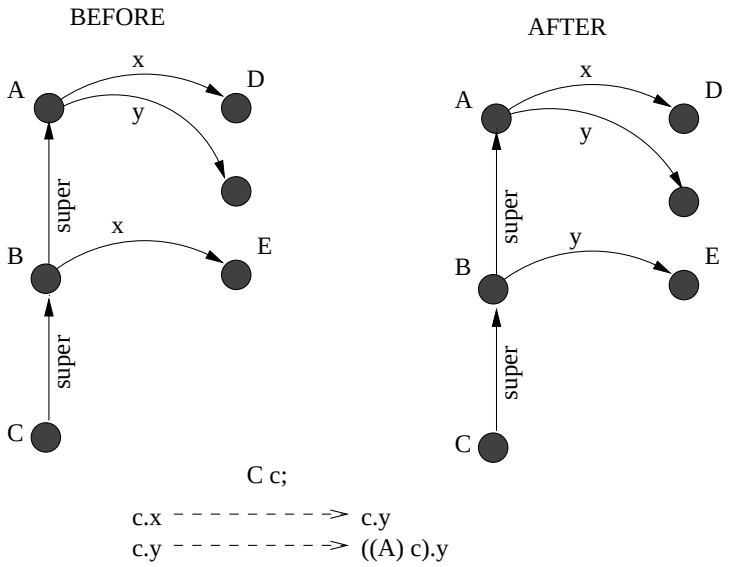

Figure 2.12: HAS-A B::x is renamed to y. This hides A::y and unhides A::x. Compensating changes to primary expressions from C are shown.

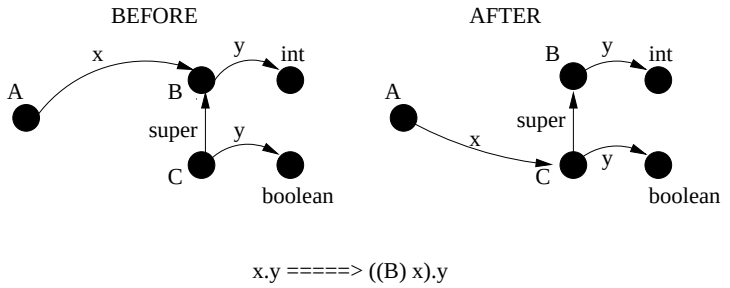

Figure 2.13: HAS-A A::x is retargeted from B to C, a narrowing of the target. The primary expression x.y is transformed to compensate.

2.6.12 REMULTIPLY HAS-A

- **Primitive:** REMULTIPLY HAS-A *has-aName* SOURCE *SourceClassName* NEW MULTIPLICITY (0 | 1)

- **Example:**
  REMULTIPLY HAS-A foo SOURCE X NEW MULTIPLICITY 1
- **Description:** Change the multiplicity of the HAS-A link to either "1" for a 1:1 link, or "0" for a 1:n link.

2.6.13 RETARGET HAS-A

- **Primitive:** RETARGET HAS-A *has-aName* SOURCE *SourceClassName* NEW TARGET *NewTargetClassName*
- **Example:** RETARGET HAS-A x SOURCE A NEW TARGET C
- **Description:** The HAS-A is retargeted. If it had an inverse HAS-A, the inverse relationship is dropped for both. Generally, a retargeting is harmful, but under special circumstances such as narrowing or widening the target, it is possible to transform primary expressions which use the HAS-A, so as to compensate. See Figure 2.13.

2.6.14 ADD OPERATION

- **Primitive:** ADD OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) TARGET *typeName*
- **Example:** ADD OPERATION foo SOURCE B PARAMETERS (int) TARGET int
- **Description:** The operation is added to class X.
- **Note:** Figure 2.14 shows the addition of an operation to class B. Note that this will likely change the behavior of call c.foo(5), where c is in class C, since it hides the old operation foo(int) inherited from class A. Unlike the situation for shadowed fields, Java does not provide a way for c to access the old foo(int) defined in A, even by casting.

2.6.15 DROP OPERATION

- **Primitive:** DROP OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...)
- **Example:** DROP OPERATION op SOURCE X PARAMETERS (p1, p2)
- **Description:** The operation is dropped. This might unhide an inherited operation.

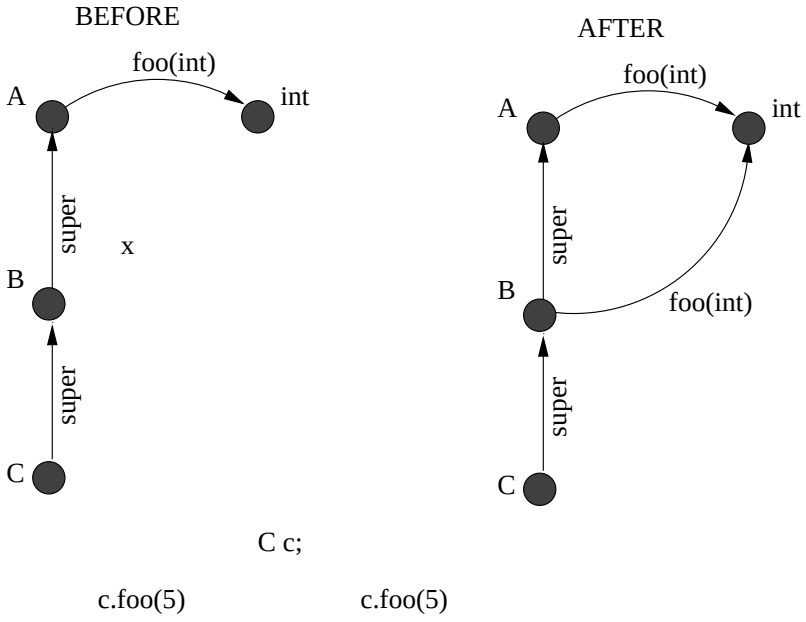

Figure 2.14: Before and after adding operation foo(int) to class B

2.6.16 RENAME OPERATION

- **Primitive:** RENAME OPERATION *oldOperationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ..) NEW NAME *newOperationName*
- **Example:** RENAME OPERATION op1 SOURCE X PARAMETERS (float, int) NEW NAME op2
- **Description:** The operation is renamed in its declaration, and in each instance of its usage in an expression.
- **Note:** If the operation's new name hides an inherited operation, the return types are required to be the same.

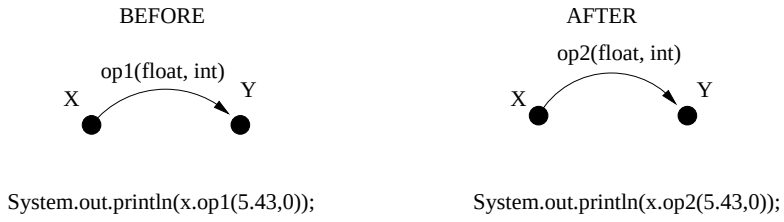

Figure 2.15: Before and after renaming an operation

2.6.17 RETARGET OPERATION

- **Primitive:** RETARGET OPERATION *operationName* SOURCE *ClassName* PARAMETERS (*type1*, *type2*, ...) NEW TARGET *newType*

- **Example:** RETARGET OPERATION foo SOURCE X PARAMETERS () NEW TARGET B
- **Description:** The return type of the operation is changed.

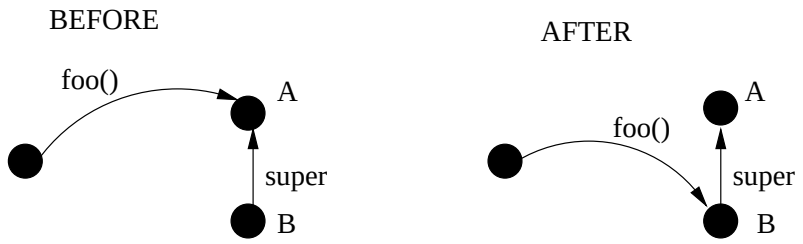

Figure 2.16: Before and after retargeting operation foo() in class X from A to B

2.6.18 ADD IS-A

- **Primitive:** ADD IS-A *SubclassName SuperclassName*
- **Example:** ADD IS-A X Y
- **Description:** An extends clause is added to the declaration of the subclass. See Figure 2.17.

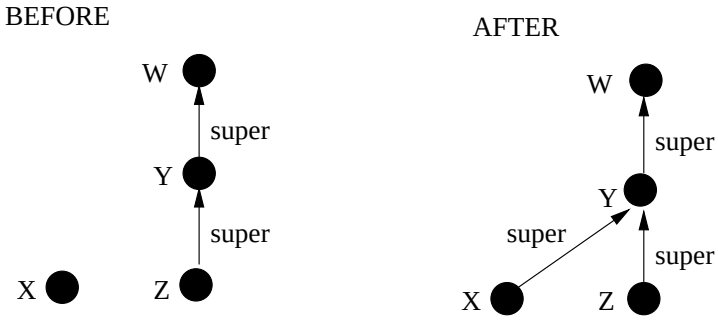

Figure 2.17: Before and after adding an IS-A link

2.6.19 DROP IS-A

- **Primitive:** DROP IS-A *SubclassName SuperclassName*
- **Example:** DROP IS-A X Y
- **Description:** The extends clause is removed from the subclass.

2.6.20 DELEGATE FIELD

- **Primitives:**
  1. DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *hasaName*
  2. DELEGATE FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*
- **Examples:**
  1. DELEGATE FIELD f SOURCE B USING HAS-A e (See Figure 2.18)
  2. DELEGATE FIELD f SOURCE B USING OPERATION getE
- **Description:** Field f is removed from class B and added to class E, where E = target(e) for HAS-A e, or E = type(getE()) for operation getE(). All primary expressions accessing f from B, are extended by adding either prefix "e." or prefix "getE()" before f. Delegate is here to support the HLP AGGREGATE.
- **Note:** Delegating is only possible along an operation which takes no parameters, in other words, a *virtual field*.

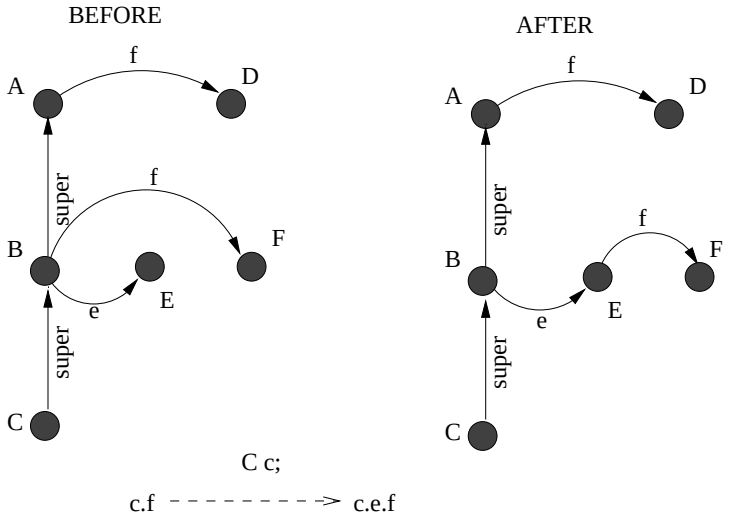

Figure 2.18: Field f is delegated using e. This unhides A::f. Compensating changes to a primary expression using f from C is shown.

2.6.21 RECLAIM FIELD

- **Primitives:**
  1. RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING HAS-A *hasaName*

2. RECLAIM FIELD *fieldName* SOURCE *SourceClassName* USING OPERATION *operationName*

- **Examples:**

1. RECLAIM FIELD f SOURCE X USING HAS-A y
2. RECLAIM FIELD f SOURCE X USING OPERATION getY

- **Description:** Field f is added to class X, and primary expressions containing accesses from X to f using "h.f", or "getY()" are reclaimed by replacing "h.f" or "getY().f' by "f". Reclaim Field supports the HLP Decompose.

2.6.22 DELEGATE OPERATION

- **Primitives:**

1. DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*
2. DELEGATE OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

- **Examples:**

1. DELEGATE OPERATION foo() SOURCE X USING HAS-A y
2. DELEGATE OPERATION foo() SOURCE X USING OPERATION getY

- **Description:** OPERATION foo is removed from class X and added to class Y, where Y = target(y) for HAS-A y, or Y = type(getY()) for operation getY(). All primary expressions accessing foo() from X, are extended by adding either prefix "y." or prefix "getY()" before foo. Delegate is here to support the HLP AGGREGATE.

2.6.23 RECLAIM OPERATION

- **Primitives:**

1. RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*
2. RECLAIM OPERATION *operationName* SOURCE *SourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*

- **Examples:**

1. RECLAIM OPERATION foo SOURCE X PARAMETERS () USING HAS-A y
2. RECLAIM OPERATION foo SOURCE X PARAMETERS () USING OPERATION getY()

- **Description:** OPERATION foo is added to class X. All primary expressions accessing foo() from X using components y.foo() or getY().foo(), are contracted by reducing the components to foo(). Reclaim is here to support the HLP DECOMPOSE.

2.6.24 LIFT FIELD

- **Primitive:** LIFT FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*
- **Example 1:** LIFT FIELD name FROM Contractor TO Staffer. (Lifting an attribute)
- **Example 2:** LIFT FIELD worksOn FROM Contractor TO Staffer. (Lifting a HAS-A)
- **Description:** Removes a field (attribute or HAS-A) from a subclass, and adds it to the superclass, if not already there. LiftField is used to support Factor.

2.6.25 LIFT OPERATION

- **Primitive:** LIFT OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*
- **Example:** LIFT OPERATION newAddress SOURCE Contractor PARAMETERS (String) TO Staffer.
- **Description:** Removes an operation from a subclass and adds it to the superclass, if not already there. LiftOperation is used to support Factor. A check is made to make sure that properties used by the operation are available in the superclass.

2.6.26 LOWER FIELD

- **Primitive:** LOWER FIELD *FieldName* SOURCE *ClassName1* TO *ClassName2*
- **Example 1:** LOWER FIELD name SOURCE Staffer TO Contractor. (Lowering an attribute)
- **Example 2:** LOWER FIELD worksOn FROM Contractor TO Staffer. (Lowering a HAS-A)
- **Description:** Copies a field (attribute or HAS-A) from a superclass to a subclass. Lower Field is used to support Flatten.

2.6.27 LOWER OPERATION

- **Primitive:** LOWER OPERATION *operationName* SOURCE *ClassName1* PARAMETERS (*type1*, *type2*, ...) TO *ClassName2*
- **Example:** LOWER OPERATION newAddress SOURCE Staffer PARAMETERS (String) TO Contractor
- **Description:** Copies an operation (and its method body) from a superclass. LowerOperation is used to support Flatten.

2.7 Higher Level Primitives (HLP's) Listed

Higher level primitives (HLP's) [23, 24, 101] are designed to succinctly express broader kinds of schema change as might be envisaged on a software engineering level. Each higher level primitive is expanded into a program of lower level primitives for execution. Using a single higher level command such as "FLATTEN A", the software designer can generate a number of lower level primitives which, taken together, will migrate fields and methods to the immediate subclasses of A.

Figure 2.19 illustrates several HLP's using OMT [97] style drawings. The full set of HLP's follows:

1. **Factor** - FACTOR $className_1$, $className_2$, ... INTO *abstractClassName*
2. **Flatten** - FLATTEN *className*
3. **Merge** - MERGE $className_1$, $className_2$, ... INTO *newClassName* USING *attributeName*
4. **Split** - SPLIT *singleClass* USING *distinguishingAttribute* VALUES (*value1, value2 ...*)
5. **Aggregate** - AGGREGATE $className_1$, $className_2$... INTO *partClassName* USING *hasaName*
6. **Decompose** - DECOMPOSE *partClassName*

The higher level primitives are intended as a convenience to the program designer. They add no power beyond what is already available using the lower level primitives. Using an HLP, the designer can, in a single operation, produce program modifications that would otherwise require a number of separate operations. However, this requires accepting the default behavior of the HLP's. For example, the Factor HLP behaves by lifting all common fields of its input classes to the superclass indicated. If, in addition, the designer is willing to forego guaranteed behavior preservation, Factor also lifts all common operations as determined by signature matching. The designer who needs to exercise greater control over which fields and operations are lifted should avoid Factor, and instead use individual Lift Field and Lift Operation primitives.

As an example of how an HLP can be expanded into a sequence of LLP's, consider Figure 2.19. The command: "FACTOR Employee, Customer INTO Person" would be expanded into:

ADD CLASS Person
ADD IS-A Customer Person
ADD IS-A Employee Person
LIFT FIELD name SOURCE Customer TO Person
LIFT FIELD name SOURCE Employee TO Person
LIFT FIELD address SOURCE Customer TO Person

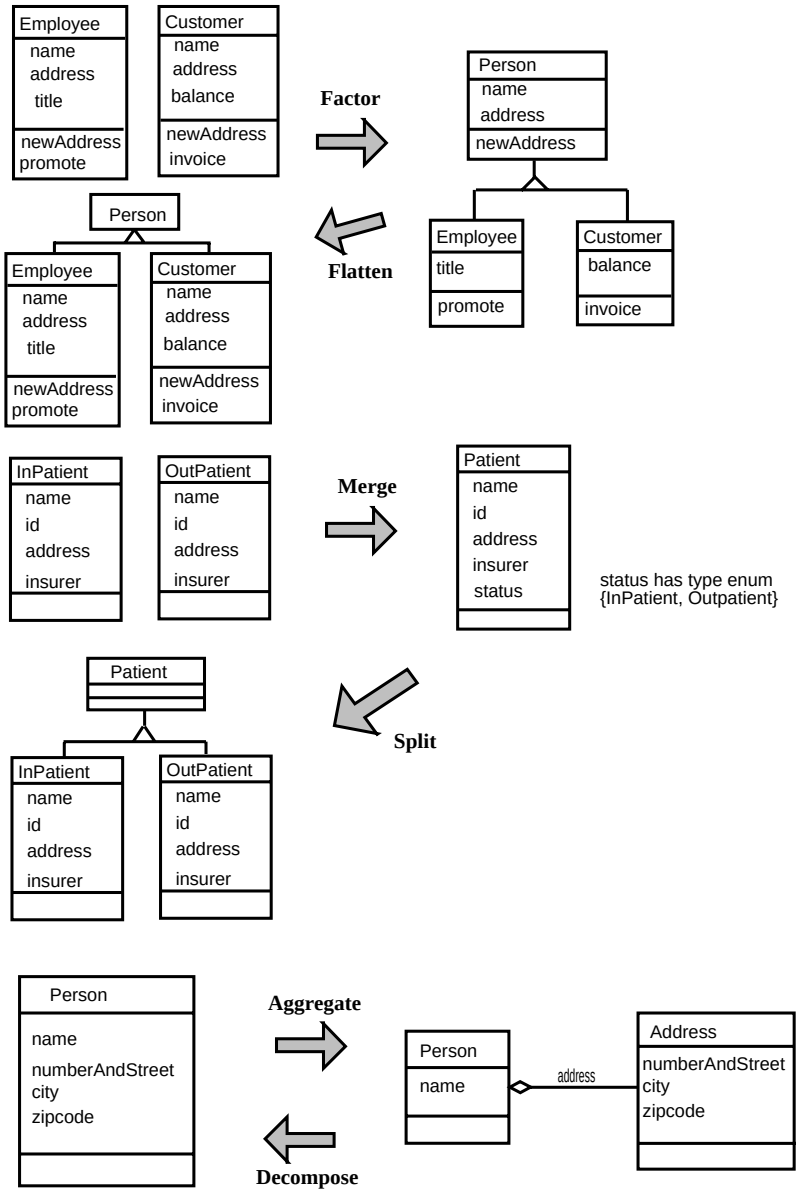

Figure 2.19: Some Higher Level Primitives Illustrated

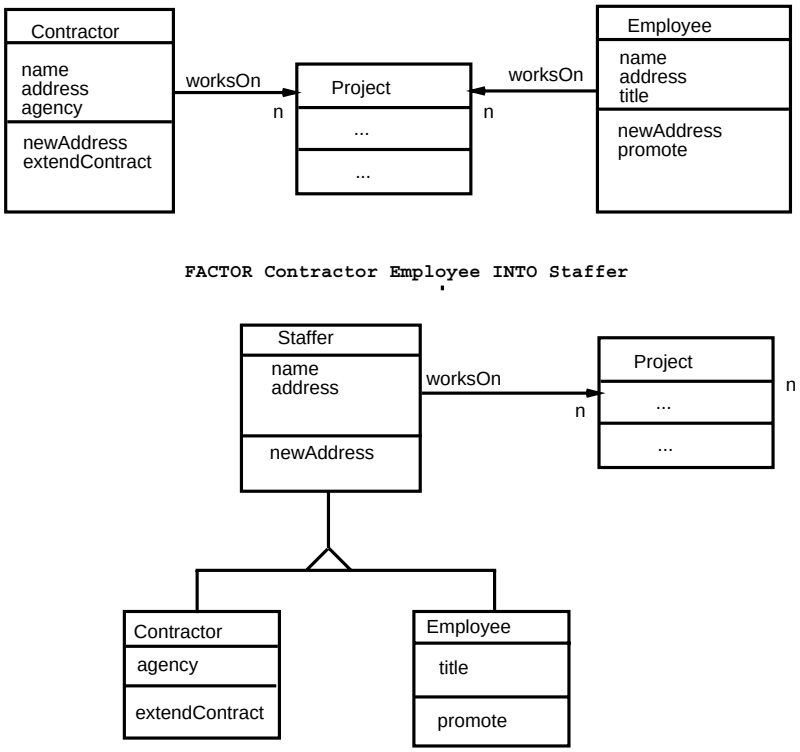

Figure 2.20: Illustrating the Effect of the Factor HLP

LIFT FIELD address SOURCE Employee TO Person
LIFT OPERATION newAddress SOURCE Customer PARAMETERS (String) TO Person
LIFT OPERATION newAddress SOURCE Employee PARAMETERS (String) TO Person

2.8 Higher Level Primitives Described

A program designer using the HLP tools must indicate whether he/she requires guarantees that the resulting system will preserve the behavior of the original, or whether guarantees of type-soundness preservation are sufficient. It is recursively undecidable to compare two method bodies and decide if they produce the same behavior. Hence, when the designer requires behavior preservation guarantees, the HLP's will not generate instructions which replace methods. However, a designer who knew that such replacements were in fact, *safe* could forego the guarantees, and the HLP's would generate the replacement instructions.

The HLP's are now described in detail. However, examination of the pre and post conditions of the HLP's and how they relate to conditions of LLP's is deferred until Section 3.8.

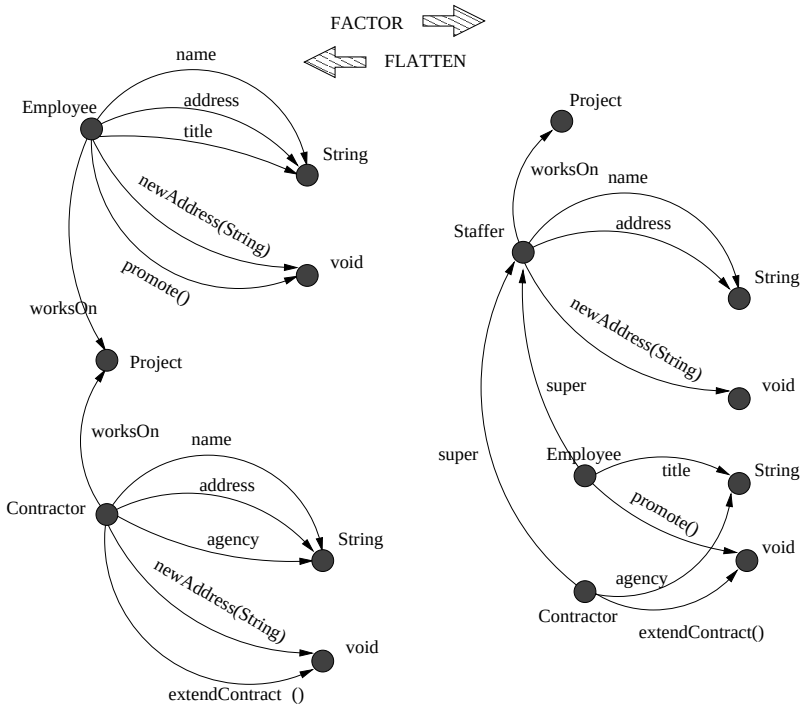

Figure 2.21: IOM Graphs showing the effects of the Factor and Flatten transformations

2.8.1 FACTOR

- **Primitive:** FACTOR $className_1$, $className_2$... INTO *abstractClassName*
- **Example:** FACTOR Contractor, Employee INTO Staffer.

  This example is illustrated in Figure 2.20 using an OMT diagram, and again in Figure 2.21 using IOM Graphs.

  In this case, the FACTOR primitive would generate the following LLP's:

  ADD CLASS Staffer
  ADD IS-A Contractor Staffer
  ADD IS-A Employee Staffer
  LIFT FIELD name SOURCE Contractor TO Staffer
  LIFT FIELD name SOURCE Employee TO Staffer
  LIFT FIELD address SOURCE Contractor TO Staffer
  LIFT FIELD address SOURCE Employee TO Staffer
  LIFT FIELD worksOn SOURCE Contractor TO Staffer
  LIFT FIELD worksOn SOURCE Employee TO Staffer
  LIFT OPERATION newAddress SOURCE Contractor PARAMETERS (String) TO Staffer
  LIFT OPERATION newAddress SOURCE Employe PARAMETERS (String) TO Staffer

- **Description:** The intent of FACTOR is to generalize the input classes by locating their common properties, and lifting them to a newly created *abstract class*. When

the user requires guaranteed behavior preservation, only the attributes and HAS-A's are lifted, otherwise matching operations are also lifted.

2.8.2 FLATTEN

- **Primitive:** FLATTEN *superClassName*
- **Example:** FLATTEN Staffer

  This example refers to Figure 2.21. The FACTOR primitive would generate the following LLP's:

  LOWER FIELD name SOURCE Staffer TO Contractor
  LOWER FIELD name SOURCE Staffer TO Employee
  LOWER FIELD address SOURCE Staffer TO Contractor
  LOWER FIELD address SOURCE Staffer TO Employee
  LOWER FIELD worksOn SOURCE Staffer TO Contractor
  LOWER FIELD worksOn SOURCE Staffer TO Employee
  LOWER OPERATION newAddress SOURCE Staffer PARAMETERS (String) TO Contractor
  LOWER OPERATION newAddress SOURCE Staffer PARAMETERS (String) TO Employee

- **Description:** The intent is to move the members of the superclass down to each of its immediate subclasses. Fields are simply copied down. Operations, including method bodies, are copied down except to subclasses where they are overridden. The superclass is retained, as well as the IS-A links from the subclasses.

2.8.3 MERGE

- **Primitive:** MERGE $className_1$, $className_2$... INTO *newClassName* USING *attributeName*
- **Example:** MERGE InPatient, OutPatient INTO Patient USING status
- **Description:** Merge takes a set of input classes which are identical in that their fields match, and their operations match up to signature. Merge creates a new class which has all the attributes, HAS-A's, operations and methods of its input classes plus a new attribute which distinguishes which input class an object came from. An enum would be the ideal type for the distinguishing attribute, but enums are not supported in Java [42], so strings comprising the names of the input classes are used instead. Merge requires the designer to forego guaranteed behavior preservation, since signature matching of operations does not imply behavior matching.

  In the example in Figure 2.19, InPatient and OutPatient have the same fields and operations. A new class Patient is created. It has the same fields and operations plus one more named status, of type String. InPatient and OutPatient are then dropped.

  In the event that the Java system describes a populated database, the old instances of both InPatient and OutPatient would need to be migrated to the Patient class. To

migrate an InPatient instance, a new Patient object would be created, its status field set to "InPatient", and the other fields copied from the InPatient object, which would then be destroyed. Similarly for OutPatient instances.

2.8.4 SPLIT

- **Primitive:** SPLIT *singleClass* USING *distinguishingAttribute* VALUES (*value1, value2* ...)
- **Examples:**
  1. SPLIT Patient USING status VALUES ("InPatient", "OutPatient")
  2. SPLIT Employee USING exempt VALUES (true, false)
  3. SPLIT Comp585 USING section VALUES (1,2,3)
- **Description:** Creates new classes from the *singleClass*, one for each value indicated for its *distinguishingAttribute*. The *singleClass* remains as a target of IS-A and HAS-A links with the new classes inheriting from it. The *distinguishingAttribute* must be either a string, or an enumerable type such as int or boolean. The names of the new classes are formed by concatenating the *singleClass*, followed by the *distinguishingAttribute*, followed by the value used, with underscores as separators.

  In the first example the new classes are named Patient_status_InPatient, and Patient_status_OutPatient. In the second example they are Employee_exempt_true, and Employee_exempt_false, and in the third example, Comp585_section_1,Comp585_section_2 and Comp585_section_3.

  If the Java system describes a populated database, the instances are distributed to the new classes according to their current values for the *distinguishingAttribute*.

2.8.5 AGGREGATE

- **Primitive:** AGGREGATE $className_1$, $className_2$... INTO *partClassName* USING *hasaName*
- **Example:** AGGREGATE Person, Company INTO Address USING address
- **Description:** The intent is to delegate common properties of the input classes into a newly created class, the *part class*. When the user requires guaranteed behavior preservation, only the attributes and HAS-A's are delegated, otherwise matching operations are also delegated.

  In the example illustrated in Figure 2.22, a new class named Address is created. The fields numberAndStreet, city and zipcode, and the operation newAddress(...) are moved to it from Person and Company. The Person and Company classes gain a HAS-A named address, with target class Address.

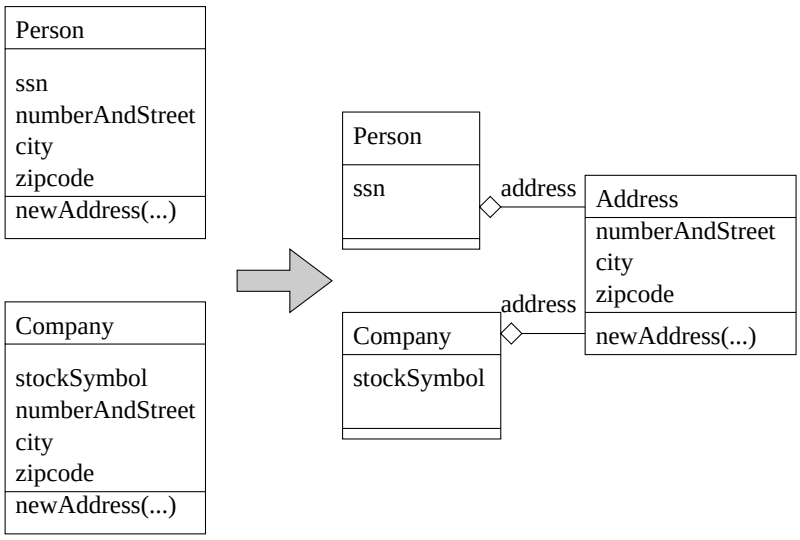

Figure 2.22: Illustrating the effect of "AGGREGATE Person, Company INTO Address USING address"

2.8.6 DECOMPOSE

- **Primitive:** DECOMPOSE *partClass*
- **Example:** DECOMPOSE Address
- **Description:** The fields and operations of the *PartClass* are distributed, using RECLAIM operations, to all classes which have a HAS-A or operation with target of *partClass*.

2.9 Comparing the CSL and IOM to Related Work

In 1987 Banerjee [11] proposed a list of possible transformations on object models. It is basically an enumeration of the transformations that result when the usual operations of adding, dropping, renaming, and altering a property are applied to the constructs in an object-oriented data model. This list was picked up by Zicari [115] who applied it for use in the O_2 database. The data model is, of course, the O_2 data model. Later on, Breche [23] and Ferrandina [24] continued Zicari's work introducing the notions of high level and low level primitives included in a change specification language. This language is drawn on here.

Although the IOM is a particularly simple object model it draws on an extensive literature in object modeling and type theory. Some early work includes [2, 31]. Recent industrial work includes that done by the ODMG [29] and the Rational group working on UML [21]. A good source for a formal treatment of object models is [66]. Cardelli [26] has produced a very readable article on type theory.

2.9.1 Java Binary Compatibility

Chapter 13 of the Java Language Specification [45], is devoted to the issue of *binary compatibility*, an issue that has also been dealt with [43] by IBM in its System Object Model (SOM). A Java library is distributed in binary form, that is, as a package of class files written in accordance with a set of properties defined in the Java Virtual Machine Specification [73]. An Application Programming Interface (API) describes the data members and methods which are provided. The designers of Java foresaw that from time to time the packages would be enhanced, and new binaries distributed. They provided for the possibility of making a number of changes to the class files, while preserving compatibility with existing programs. To quote from [45], *A change to a type is binary compatible with ... preexisting binaries if preexisting binaries that previously linked without error will continue to link without error.*

To achieve binary compatibility the rules in [73] require, among other things, that references to data members use their name, rather than, say, an offset from the object location. According to [45], a number of binary compatible updates are allowed including:

- Reimplementing existing methods to improve performance fix bugs, or add new features.
- Adding new fields, methods, or constructors to an existing class or interface.
- Reordering the fields, methods or constructors in an existing type declaration.
- Moving a method upward in the class hierarchy, provided a forwarding method is left in its place.

In [45] it is stressed that binary compatibility is not the same as source compatibility. In fact, it may be the case that if a set of source files is compiled together, and then one of them evolves in a binary compatible way, they will no longer be compilable together. It is preferable to evolve the source files using a tool such as CSL, rather than rely on binary compatibility. It should also be mentioned that CSL supports a number of transformations that go beyond binary compatibility, and that although the present work prototypes CSL transformations to Java code, there is no reason not to apply transformations to C++ or other languages which do not support binary compatibility.

2.9.2 Work done by the Demeter Group in Adaptive Programming

This section compares the present research to earlier work done within the Demeter group by Paul Bergstein [14, 15], Walter Hürsch [16, 55, 54], and Linda Seiter [55]. There is a great deal of overlap in the transformations proposed in all this research as would be expected since they all are variants or extensions of transformations proposed earlier by Banerjee [11] and Zicari [115].

The CSL transformations will be found to differ from others proposed in the Demeter group in that they address a somewhat different data model, one more in conformance to industry

practices. Also, some CSL higher level primitives go beyond the other primitives proposed, and CSL transformations are expressed in a SQL-like language. CSL transformations are validated by working tools which have been tested on Java language programs.

Data Model Differences

Bergstein in his thesis [15] refers to the so-called *class dictionary graph* [72] used as part of the Demeter data model. It consists of vertices and edges. The vertices are either *alternation vertices* which roughly correspond to abstract classes, i.e. they are never instantiated, or *concrete vertices* which may be instantiated but are never inherited from. The edges are either unlabelled *alternation edges* which indicate inheritance by pointing from an alternation class to a subclass, or labelled *construction edges* which support references from one object to another. The requirement that alternation classes can never be instantiated and that concrete classes can never be inherited from is known as the *abstract superclass rule.*

This model differs in several important ways from the IOM proposed here.

1. The abstract superclass rule is not present in the IOM. Nor is it present in either the UML or ODMG object models, nor in either C++ or Java.
2. IOM allows HAS-A links to be multiple valued while the reference links in Bernstein's model are single valued. Multiple valued associations are allowed by both ODMG and UML. They are easily implemented in C++ using template classes. They can presently be implemented in Java using arrays and collection classes such as Vector. Currently there are research proposals [80, 77, 3] for extending templates into Java as well.
3. IOM HAS-A links are stored with the names of optional inverse HAS-A links. This is in accordance with ODMG practice and makes it easier to identify HAS-A links as being parts of the same Design View association.
4. Operations are not explicitly shown in a class dictionary graph. This is in accordance with the principle of separating behavior from structure. The IOM, on the other hand, does model operation signatures and the CSL allows manipulation of operation names, parameters and return types.

Hürsch [54] refers to a *kernel* data model which differs from the class dictionary graph in that vertices can represent the primitive concrete classes Integer, Real, String and Boolean as well as user-defined concrete classes and abstract classes. Also, as in the IOM, method signatures form part of the model. He retains the abstract superclass rule.

Differences Among the Transformations

As ably pointed out by Hürsch [54] the number and type of primitive transformations needed is directly determined by the constructs used in the data model. He calls the set {add, delete, change, rename} the *meta primitive transformations.* The add and delete meta

primitive transformations can be applied to any of the data model constructs, the rename to any labelled construct, and the change to any construct capable of being changed. It follows that the fuller the data model, the more primitive transformations there will be.

The IOM data model has several features lacking in either class dictionary graphs or the kernel model, namely primitive valued attributes, multiplicities for references, and operation signatures. It follows that the set of primitive transformations is richer.

Bergstein uses the term *object-equivalent* to describe two class dictionary graphs which allow the same set of objects. A class dictionary graph ϕ_2 is said to *extend* class dictionary graph ϕ_1 if loosely speaking, it allows all objects allowed by ϕ_1. *Object-preserving* transformations to a class graph produce an object-equivalent class graph, while *object-extending* transformations produce a class graph which extends the original.

Bergstein considers 5 object-preserving transformations:

1. Deletion of useless alternation (DUA)
2. Addition of useless alternation (AUA)
3. Abstraction of common parts (ACP)
4. Distribution of common parts (DCP)
5. Part Replacement (PRP)

and 3 object-extending ones:

1. Class Addition (CAD)
2. Part Addition (PAD)
3. Part Generalization (PGN)

Of these, DUA and AUA are direct results of the abstract superclass rule, and correspond to special cases of the CSL transformations ADD CLASS and DROP CLASS. ACP and DCP are similar to FACTOR and FLATTEN respectively, which are higher level primitives in CSL. PRP and PGN could be implemented in CSL by RETARGET HAS-A and LIFT FIELD, respectively,, and PAD by ADD HAS-A.

Hürsch distinguishes between *basic* primitive class graph transformations and *composite* ones. This is similar to the separation of CSL into LLP and HLP commands. In common with CSL, but unlike Bergstein, Hürsch includes potentially destructive commands such as deleting a reference.

Hürsch's basic primitive class graph transformations are:

- Addition of concrete class (AddC)

- Deletion of concrete class (DelC)
- Renaming of class (RenC)
- Addition of abstract class (AddA)
- Deletion of abstract class (DelA)
- Addition of reference relation (AddR)
- Deletion of reference relation (DelR)
- Renaming of Reference (RenR)
- Addition of inheritance relation (AddI)
- Deletion of inheritance relation (DelI)
- Replacement of reference relation(RepR)
- Generalization of reference relation (GenR)

The composite transformations are:

- Addition of Subclass (AddS)
- Abstraction of Common Reference (AbsR)
- Distribution of Common Reference (DisR)
- Telescoping of Reference (TelR)
- Telescoping of Inheritance (TelI)

Figure 2.23 compares Bergstein's transformations with the Kernel model transformations of Hürsch and Seiter and the CSL transformations.

It should be noted that CSL allows transformations that are neither object preserving nor object extending. In such cases no guarantees can be made for preserving behavior, or for that matter data. One reason for this is to enable higher level primitives to be expanded into sequences of lower level ones which when executed together conserve behavior even though they don't conserve it when executed in isolation.

Another area of difference between CSL and the other Demeter research on transformations is the target language for the transformations. Bergstein targeted the language of Adaptive Programming with class dictionary graphs and propagation patterns. Hürsch and Seiter likewise considered mainly adaptive programs and were able to obtain proofs of consistency and behavior maintenance for transformations on them. Hürsch additionally considered transformations on CLOS and C++ programs but more in the mode of discussion than actual proofs.

Bergstein	**Kernel** **Hürsch and Seiter**	**CSL** **Werner**
CAD	AddC	ADD CLASS
AUA	AddA	ADD CLASS
	DelC	DROP CLASS
DUA	DelA	DROP CLASS
	RenC	RENAME CLASS
PAD	AddR	ADD HAS-A
	DelR	DROP HAS-A
	RenR	RENAME HAS-A
	AddI	ADD IS-A
	DelI	DROP IS-A
PRP	RepR	RETARGET HAS-A
PGN	GenR	LIFT FIELD
	AddS	
ACP	AbsR	FACTOR
DCP	DisR	FLATTEN
	TelR	DELEGATE FIELD
	TelI	
		ADD ATTRIBUTE
		DROP ATTRIBUTE
		RENAME ATTRIBUTE
		RETARGET ATTRIBUTE
		REMULTIPLY ATTRIBUTE
		ADD OPERATION
		RENAME OPERATION
		RETARGET OPERATION
		DROP OPERATION
		REMULTIPLY HAS-A
		MERGE
		SPLIT
		AGGREGATE
		DECOMPOSE

Figure 2.23: Comparing Transformations - Bergstein - Kernel - CSL

2.9.3 Refactorings

The body of work identified as *refactoring* is associated with Ralph Johnson [59], his student William Opdyke [81], and John Brant [95], all at the University of Illinois. A refactoring is defined as a *behavior preserving transformation*. Opdyke defines it this way, referring to C++ programs: "If the function *main* is called twice (once before and once after a refactoring) with the same set of inputs, the resulting set of output values must be the same." Twenty six low-level, and several high-level refactorings are described. There is considerable overlap to the primitives described here, with the difference that refactorings are stated as functions, rather than in a command language form.

Opdyke's thesis targets a slightly simplified C++, for example, no overloaded member function names, and no casts. No additional data modeling is done, which means that preconditions are stated in terms of C++ constructs which may include things like global variables, non-member functions, and so on. There is no visual representation of the programs, other than OMT. Fields which are class members are not always clearly distinguished from variables which are defined inside method bodies. Opdyke includes a few primitives beyond what is available in CSL, particularly as regards the manipulation of the contents of method bodies, however these depend on checking preconditions using predicates such as:

semanticallyEquivalentP((function or statement) F1, (function or statement) F2);

which, in the general case are uncomputable.

Contrasting CSL with refactoring:

- CSL targets a *tighter* language, namely Java.
- CSL commands reference the IOM data model with its visual representation.
- CSL distinguishes between type-soundness preservation and behavior preservation, allowing the user of the tool to choose either.
- CSL preconditions are clearly stated in Chapter 3 in terms of both the IOM constructs, and usage trace mappings from expressions found inside method bodies to the constructs which they use.

2.10 The EBNF For CSL Commands

CslCommand := LlCommand | HlCommand

2.10.1 The Lower Level Primitives

- LlCommand := (AddCommand | DropCommand | RenameCommand | RetargetCommand | RemultiplyCommand | DelegateCommand | ReclaimCommand | LiftCommand | LowerCommand)

- AddCommand := “ADD” (AddClass | AddAttribute | AddHasa | AddOperation | AddIsa)
- DropCommand := “DROP” (DropClass | DropAttribute | DropHasa | DropOperation | DropIsa)
- RenameCommand := “RENAME” (RenameClass | RenameAttribute | RenameHasa | RenameOperation)
- RetargetCommand := “RETARGET” (RetargetAttribute | RetargetHasa | RetargetOperation)
- RemultiplyCommand := “REMULTIPLY” (RemultiplyAttribute | RemultiplyHasa)
- DelegateCommand := “DELEGATE” (DelegateAttribute | DelegateHasa | DelegateOperation)
- ReclaimCommand := “RECLAIM” (ReclaimAttribute | ReclaimHasa | ReclaimOperation)
- LiftCommand := “LIFT” (LiftField | LiftOperation)
- LowerCommand := “LOWER” (LowerField | LowerOperation)
- AddClass := “CLASS” Identifier
- AddAttribute:= “ATTRIBUTE” FieldTag “TARGET” Identifier
- AddHasa := “HAS-A” FieldTag “TARGET” Identifier “MULTIPLICITY” <INTEGER_LITERAL> [“INVERSE” Identifier]
- AddOperation:= “OPERATION” OperationTag “TARGET” Identifier
- AddIsa := “IS-A” Identifier Identifier
- DropClass := “CLASS” Identifier
- DropAttribute:= “ATTRIBUTE” FieldTag
- DropHasa := “HAS-A” FieldTag
- DropOperation:= “OPERATION” OperationTag
- DropIsa := “IS-A” Identifier Identifier
- RenameClass:= “CLASS” Identifier “NEW” “NAME” Identifier
- RenameAttribute:= “ATTRIBUTE” FieldTag “NEW” “NAME” Identifier
- RenameHasa:= “HAS-A” FieldTag “NEW” “NAME” Identifier
- RenameOperation:= “OPERATION” OperationTag “NEW” “NAME” Identifier
- RetargetAttribute:= “ATTRIBUTE” FieldTag “NEW” “TARGET” Identifier

- RetargetHasa:= “HAS-A” FieldTag “NEW” “TARGET” Identifier
- RetargetOperation:= “OPERATION” OperationTag “NEW” “TARGET” Identifier
- RemultiplyAttribute:= “ATTRIBUTE” FieldTag “NEW” “MULTIPLICITY” <INTEGER_LITERAL>
- RemultiplyHasa:= “HAS-A” FieldTag “NEW” “MULTIPLICITY” <INTEGER_LITERAL>
- DelegateField:= “FIELD” FieldTag “USING” (UsingHasa | UsingOperation)
- DelegateOperation:= “OPERATION” OperationTag “USING” (UsingHasa | UsingOperation)
- ReclaimField:= “FIELD” FieldTag “USING” (UsingHasa | UsingOperation)
- ReclaimOperation:= “OPERATION” OperationTag “USING” (UsingHasa | UsingOperation)
- LiftField := “FIELD” FieldTag “TO” Identifier
- LiftOperation := “OPERATION” OperationTag “TO” Identifier
- LowerField := “FIELD” FieldTag “TO” Identifier
- LowerOperation := “OPERATION” OperationTag “TO” Identifier
- UsingHasa := “HAS-A” FieldTag
- UsingOperation:= “OPERATION” FieldTag
- FieldTag := Identifier “SOURCE” Identifier
- OperationTag := FieldTag “PARAMETERS” “(” (Identifier)* “)”
- Identifier := <IDENTIFIER>

2.10.2 The Higher Level Primitives

- HlCommand := (FactorCommand | FlattenCommand | MergeCommand | SplitCommand | AggregateCommand | DecomposeCommand)
- FactorCommand := “FACTOR” Identifier (“,” Identifier)* “INTO” Identifier
- FlattenCommand := “FLATTEN” Identifier
- MergeCommand := “MERGE” Identifier (“,” Identifier)* “INTO” Identifier “USING” Identifier
- SplitCommand := “SPLIT” Identifier “USING” Identifier “VALUES” “(” (Identifier)* “)”
- AggregateCommand := “AGGREGATE” Identifier (“,” Identifier)* “INTO” Identifier “USING” Identifier
- DecomposeCommand := “DECOMPOSE” Identifier

Chapter 3

Preserving Type Soundness and Behavior

3.1 Introduction

The transformations described in Chapter 2 have the goal of maintaining the integrity of a Java program in the event of schema change. When the word *program* is used here, what is meant is the set of source Java files which together with imported files are used to generate the class files which comprise an application. Although the present research applies specifically to programs written in Java there is no inherent reason why the results cannot be extended to programs written in other object-oriented programming languages as well.

A prototype system demonstrates in a practical way the feasibility of the transformations. By exercising the prototype with a variety of input programs it is possible to lend credence to the validity of the transformations. However, proving validity is more difficult to do with a production language like Java than it would be with a simplified toy language.

In considering whether a proposed CSL transformation is sound, one has to ask these questions.

1. Will the transformation break the language rules of Java? In particular, will it break the type-soundness of an existing program, preventing its re-compilation?
2. Will it alter the behavior of a program so that, for example, a different output would be obtained from the same input?

We say that a transformation is *type-sound* if it respects the language rules of Java, and results in a compilable system, provided the original system was compilable. It is *behavior-preserving* if it is guaranteed to produce the same outputs as before, from the same inputs. Figure 3.1 classifies the CSL transformations.

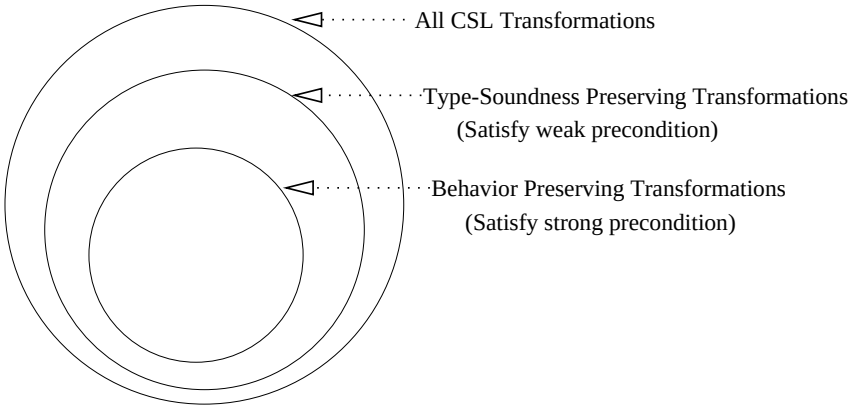

Figure 3.1: Classifying CSL Transformations

Associated with each CSL transformation are *preconditions*. Before the transformation is carried out, the preconditions are checked so as to assure the soundness of the resultant program. The checking is done using the existing program declarations and their usages in expressions. There are two types of preconditions:

1. *Weak* preconditions protect the type-soundness of the program, preserving compilability;
2. *Strong* preconditions guard against possible behavior changes that may result even though the weak preconditions are satisfied.

When programs are transformed, the user can specify which kind of preconditions are to be checked. For example, a user might waive strong precondition checking, relying on her own knowledge of the program to be able to protect against undesirable behavior changes. Another situation is when an HLP is used to generate a stream of LLP's. The HLP, knowing how the LLP's will interact, may require strong precondition checking for some, and weak for others.

Section 3.8 contains an exhaustive examination of LLP's, including weak preconditions for each, and strong preconditions for some. Additionally, compensating changes to expressions which use the altered declarations are stated. Section 3.9 does the same for HLP's, and also indicates how HLP's are expanded into sequences of LLP's. The HLP's are intended to be *correct* in the sense that if their preconditions are satisfied, the preconditions of sufficiently many spawned LLP's are satisfied so as to engender the postconditions of the HLP's.

An example will clarify some of the issues surrounding weak and strong preconditions. Consider the transformation:

ADD OPERATION op SOURCE X PARAMETERS (p1, p2) TARGET t

This will create an operation of the form `X::op(p1, p2):t` where the notation :t means that the return type is t. The weak precondition requires that class X as well as types t, p1 and p2 must already exist. Also X does not already have an operation of the form `X::op(p1, p2)`, and if the new operation hides an inherited one, `Y::op(p1, p2):t'`, where

X$\stackrel{\star}{\Longleftarrow}$Y, then t = t'. (This is due to the Java language rule that an overriding operation may not alter the return type).

If the weak precondition is satisfied, then the transformation will succeed in preserving the type-soundness of existing programs. In particular, if program P contains variable x of type X, and if the invocation `x.op(v1,v2)` where type(v1)= p1 and type(v2)= p2, would have succeeded before the transformation, then it will succeed afterwards. However, the behavior of program P may change drastically as a result of the transformation, since `X::op(p1, p2)` may return a different value and have different effects on the environment than `Y::op(p1, p2)`. To preserve behavior requires a stronger precondition, namely that the added operation does not hide an inherited one; for this example this means $op(p1, p2) \notin Operations(X)$.

The remainder of this chapter is as follows. Section 3.2 describes a subset of the Java language that will be used here. Section 3.3 extends IOM Graphs to include local variables, imported classes, and allocations. Section 3.4 maps paths in Extended IOM Graphs to possible expressions, and Section 3.5 reverses this to map expressions found in programs to paths. These mappings are summarized in Section 3.6. Considerations of the soundness of transformations are further discussed in Section 3.7. As earlier mentioned, Sections 3.8 and 3.9 examine the LLP's and HLP's in detail. Finally, Section 3.10 discusses some related work.

3.2 Java Language Subset Used

This chapter is concerned primarily with studying the effect that changes to the class hierarchy, instance fields, and method headers have on the soundness and behavior of the programs which contain their definitions. Although Java continues to evolve, these features were already fully developed in Java 1.02, which is well documented in the Java Language Specification [45]. Accordingly, the Java referred to in this chapter is a simplified version of Java 1.02. It does not include features such as reflection and inner classes, which came in later. Another simplifying assumption concerns the access modifiers of Java. Java has four, namely *private*, *public*, *protected* and none (package scope). In analyzing the effects of transformations, this produces seemingly endless combinations of cases to consider. Accordingly, to simplify the discussion that follows, it is assumed that only the *public* and *protected* modifiers are used, or if no modifier is used, all classes which are defined in the program belong to the same package. Another simplifying assumption is that neither classes nor individual members are declared *final*. It should also be made clear that the members which are referred to in CSL commands, are always instance members, not members marked *static*.

3.3 Extended IOM Graphs

The IOM Graphs introduced in Chapter 2 are adequate to describe all of the CSL transformations in terms of alterations to their nodes, arcs and labels. However, in this chapter we examine the expressions which may be affected by those transformations. Constructs such

as local variables, imported classes and allocations, are not shown in IOM Graphs since they cannot be altered by CSL commands, but they often enter into expressions. Accordingly, we extend IOM Graphs with the goal of being able to identify all primary and cast expressions found in class definitions with paths in the extended graphs.

A new type of arc, called an *initial arc* is introduced, shown in diagrams as a dashed arrow. An initial arc, as indicated by its name, can only be used at the beginning of a path.

The following extensions are made to an IOM Graph to produce an *Extended IOM Graph*:

1. Imported classes - A node is added for each imported class referred to in the program. This includes implicitly imported classes in java.lang, as well as explicitly imported classes, whether individually imported, or imported using wild cards such as java.util.*, and then referred to. In addition, arcs are added for each of the members of these classes which are used.

2. *Cast* arcs - If class B is a descendent of class A, (B $\stackrel{\star}{\Longleftarrow}$A), an *upcast* arc labelled "(A)" is added from B to A, and a *downcast* arc labelled "(B)" is added from A to B. These cast arcs are said to be *induced* by the chain of "super" arcs which connects from B to A. Cast arcs between primitive nodes are added, as allowed by the rules of Java. Note that *identity* casts, (between a class and itself), were already present in IOM Graphs. To avoid clutter, cast arcs are not normally drawn.

3. *This* arcs - An initial arc labelled "this" is added pointing to each reference node. To avoid clutter, *this* arcs are not usually drawn.

4. *Local* arcs - An initial arc is added for each local variable or parameter declaration that appears in a class definition. Its source is the Ref node for the class which contains the definition, and its target is the type of the variable or parameter. It is labelled with the variable or parameter name. As a consequence, several initial arcs with the same label may originate at a Ref node, since variables with the same name may be declared in different scopes.

5. *Allocation* arcs - An initial arc is added for each class instance creation and array allocation expression. Its source is the Ref node whose class definition contains the expression, and its target is the node representing the type of the class instance or the component type of the array. The allocation arc is labelled "new X()" if it is a class instance creation of type X, or "new a[]" for creation of an array named a. As with local variables, several allocation arcs with the same label may emanate from a Ref node.

Also, "super" arcs are considered to be initial arcs in Extended IOM Graphs, and are shown as dashed arrows. Extended IOM arcs are summarized in Figure 3.2.

Figure 3.3 illustrates an Extended IOM Graph using the TestFoo program shown in Figure 3.4.

Arc Type	Labelling	Joining	Placement
IS-A	"super"	Ref $\longrightarrow$ Ref	initial
Identity	(*node name*)	Node $\longrightarrow$ Node	
HAS-A	name	Ref $\longrightarrow$ Ref	
Attribute	name	Ref $\longrightarrow$ Prim	
RefOp	signature	Ref $\longrightarrow$ Ref	
PrimOp	signature	Ref $\longrightarrow$ Prim	
VoidOp	signature	Ref $\longrightarrow$ Void	
Cast	(*cast-to node*)	Ref $\longrightarrow$ Ref	
This	"this"	Ref $\longrightarrow$ Ref	initial
Local	variable name	Ref $\longrightarrow$ Ref	initial
Allocation	new *class name*()	Ref $\longrightarrow$ Ref	initial
Allocation	new *array name*[]	Ref $\longrightarrow$ Ref	initial

Figure 3.2: Arcs in Extended IOM Graphs

3.3.1 IOM Paths

The arcs in an Extended IOM Graph can be partitioned into sets, depending on the kind of node of an arc's source and target. So, for example, a Ref-Prim arc is one whose source is a reference node, and whose target is a primitive node. For this purpose, the "void" node will be considered as primitive. The arc kinds are:

- Ref-Ref - Source and target are both references.
- Ref-Prim - Source is a reference, target is a primitive;
- Prim-Prim - Source and target are both primitive (the only Prim-Prim arcs are the identify arcs for each primitive node);

A path in a graph is a sequence of arcs: $a_1, a_2, ..., a_n$, where target(a_i) = source(a_{i+1}), i = 1,2,3, ... n-1. An *IOM path* is a sequence of arcs: $a_1, a_2, ..., a_n$, forming a path in an extended IOM Graph, such that a_1 is an initial arc, and if a_i is followed by a_{i+1}, then a_i must be a Ref-Ref, and a_{i+1} a Ref-Ref or a Ref-Prim.

Two IOM Paths $a_1, a_2, ..., a_n$, and $b_1, b_2, ..., b_m$ can be composed, provided target(a_n) = source(b_1), and b_1 is a this arc, to form the composite arc $a_1, a_2, ..., a_n, b_2, ..., b_m$.

In Section 3.4, we will show that IOM paths represent possible primary and cast expressions that are found in method bodies. In Section 3.5, we show that all such expressions (except for literals), can be represented this way.

3.4 Mapping IOM Paths to Possible Primary Expressions

Expressions are found inside method and constructor bodies, or as initializers for variables and fields. In Java, the building blocks of expressions are the primary expressions. They

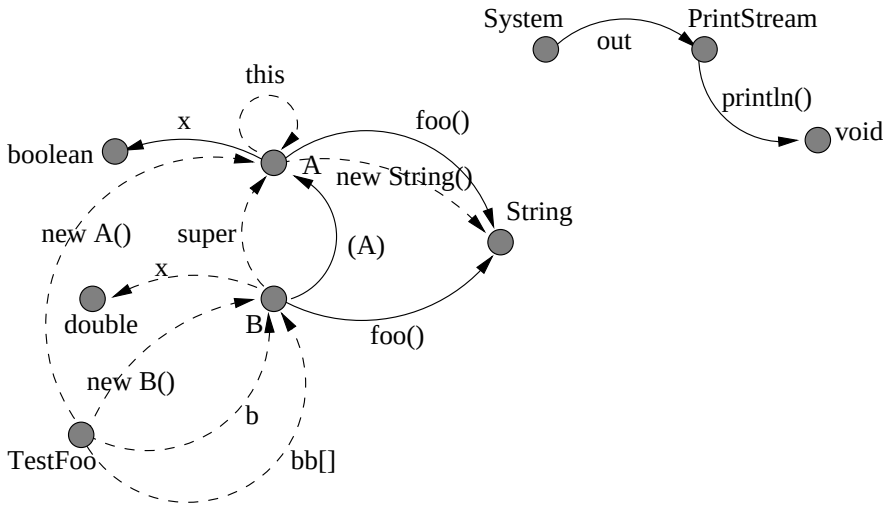

Figure 3.3: An Extended IOM Graph for the TestFoo program. Initial arcs are shown dashed.

consist of literals, "this", field accesses, method invocations, array and instance creations, array accesses, and also parenthesized expressions. All other expressions, such as relational expressions, conditional expressions, cast expressions and assignment expressions, can be built up from primary expressions using unary and binary operators and casts. A primary expression may contain periods, as in a.b.foo(5). Such primary expressions are called here *compound*, while the expression segments between periods are called its *components*. A component is considered to be *simple*.

The CSL transformations can directly affect primary and cast expressions, and indirectly affect most others. It suffices to study the primary and cast expressions. We now show how to generate such expressions from IOM Paths. Generally speaking, an arc in an Extended IOM Graph generates a simple expression, while a path of such arcs generates a compound expression.

However, certain Extended IOM Graph arcs generate multiple simple expressions:

- Arrays - An arc of the form $\mathrm{A} \xrightarrow{b[]} \mathrm{B}$, generates a family of possible expressions, $\{b[i] \mid i = 0, 1, 2, ...\}$. Note that array checking is done at run-time, not compile-time.

- Operations with parameters - An arc of the form $\mathrm{A} \xrightarrow{foo(int,boolean)} \mathrm{T}$, generates a family of possible expressions, one for each combination of values that can be substituted as actual parameters for the formal parameters of types int and boolean.

As an example of an IOM Path generating an expression, the path: $\mathrm{TestFoo} \xrightarrow{bb[]} \mathrm{B} \xrightarrow{foo()} \mathrm{String}$ in Figure 3.3, generates the expression: `bb[0].foo()`, among many others. Note that this *possible* expression does not appear in the TestFoo program, (see Figure 3.4) used to generate the graph.

```
import java.io.*;

public class TestFoo{
    public static void main(String argv[]){
        System.out.println("(new A()).x = "+(new A()).x);
        System.out.println("(new A()).foo() = "+(new A()).foo());
        B b = new B();
        System.out.println("b.foo() = "+b.foo());
        System.out.println("b.x = "+b.x);
        System.out.println("((A) b).x = "+ ((A) b).x);
        System.out.println("((A) b).foo() = "+ ((A) b).foo());
        B[] bb = new B[4];
        bb[0] = new B();
        System.out.println("bb[0].x = "+ bb[0].x);}
}
class A{
    boolean x = false;
    String foo(){System.out.println("this.x = "+this.x);
        return new String(" from A");}
}
class B extends A{
    double x = 3.14;
    String foo(){return new String(" from B");}
}
```

Figure 3.4: The TestFoo program showing various primary expression usages

If ip is an IOM Path, then *possibleUsages(ip)* is the set of expressions generated by it. The paths in possibleUsages(ip) are valid only in certain lexical scopes, however for a particular IOM Path ip, there is at least one scope block for which an expression in possibleUsages(ip) is valid. For example, the expression `bb[0].foo()` can only occur in places which fall within the static scope of an array variable bb whose component type is a reference to a class which supports a method foo(). The existence of the path: $TestFoo \xrightarrow{bb[]} B \xrightarrow{foo()} String$ indicates that there is at least one method body in class TestFoo, in which an array variable bb is in scope with component type B, and also that B includes a method foo().

IOM Paths may contain cast arcs. The expressions generated by such paths need some massaging to make them acceptable Java expressions. For example, the path:

$$TestFoo \xrightarrow{b} B \xrightarrow{(A)} A \xrightarrow{foo()} String$$

in Figure 3.3, matches the expression: `((A) b).foo()`. More generally, if a_m is a cast arc labelled "(X)", then the IOM Path: $a_1, a_2, ...a_{m-1}, a_m, a_{m+1}, ...a_n$, generates the expression:

$$((X)a_1.a_2....a_{m-1}).a_{m+1}....a_n$$

after the other a_i have been suitably expanded.

3.5 Usage Traces - Mapping Actual Primary Expressions to IOM Paths

A central issue in deciding whether it is sound to rename, alter or drop an existing IOM construct, is to determine exactly where in the program the construct is used. To do this, we introduce *usage traces*, which match primary and cast expressions found in the program to paths in Extended IOM Graphs. Due to dynamic dispatch, this needs to be done on two levels, namely:

1. compile-time (static) path matching;
2. run-time (dynamic) path matching.

Consider this example:

```
class A{
    public D foo(){ ... }
}
class B extends A{
    public D foo(){ ... }
}
class C{
    public D bix(A a){return a.foo();}
}
class D{ ... }
```

Look at the call to `a.foo()` in the method body of `C::bix(A a)`. In the formal parameter list, a was declared as a parameter of type A. Hence, A is its *static* type. However, at run-time, when the call to `a.foo()` is made, the actual version of `foo()` which is dispatched depends on the *base* class of a, which may be a subclass of A, say B.

So, the primary expression `a.foo()` corresponds to the compile-time path: $\xrightarrow{a}$ A$\xrightarrow{foo()}$ D, but at run-time to a family of paths:

$$\{\xrightarrow{a}A\xrightarrow{(X)}X\xrightarrow{foo()}Y \mid X\overset{\star}{\Longleftarrow}A, Y\overset{\star}{\Longleftarrow}D\}$$

This is shown in Figure 3.5.

A *PrimCast* expression is the term we will use for any Java expression which can be built from primary expressions and casts. In Section 3.4 it was shown that an IOM Path, ip can be used to generate a set of possible PrimCast expressions, possibleUsages(ip). However, not all such paths represent actual expressions used in the program. Knowing which paths are actually used is often of crucial importance in deciding whether an IOM construct can safely be dropped or altered. In this section we indicate how to trace PrimCast expressions along IOM Paths. In Chapter 4, it will be shown how to physically represent these traces

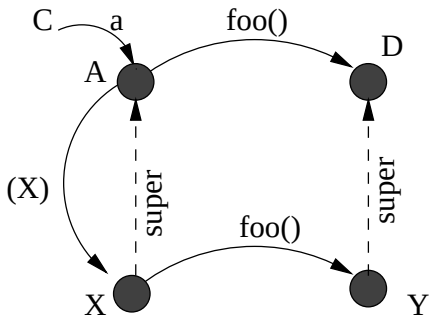

Figure 3.5: The dynamic dispatch of the call to `a.foo()`, where a has static type A, could result in traversing either of the `foo()` arcs in the figure.

in terms of pointers between data structures representing primary expression components and IOM constructs.

If expression e∈ possibleUsages(ip), for some IOM Path ip, then e can be shown graphically as a trace along ip; we call this mapping a *usage trace*. Figure 3.6 shows a usage trace along an IOM Path.

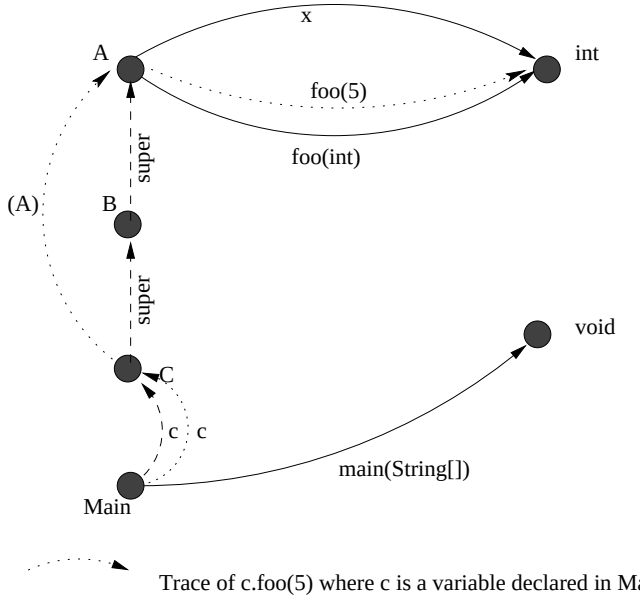

Figure 3.6: Usage trace of the primary expression `c.foo(5)` to the IOM Path: Main$\xrightarrow{c}$ C$\xrightarrow{(A)}$A$\xrightarrow{foo(int)}$ int.

For program P, PrimCast(P) is the set of *actual* PrimCast expressions found in P. We use the notation A::e to indicate that a PrimCast expression e is located in the definition of class A. Given a PrimCast expression, we need to be able to trace it along an IOM Path. Casts are easily dealt with. If x is a reference to an object of class X, then the cast: (Y) x is traced along the cast arc labelled "(Y)" leading from X to Y. Assuming the program is compilable, such an arc must exist, since either X is descended from Y or vice-versa.

Now consider Java primary expressions. According to [45], a primary expression is either a:

- Literal;
- this;
- (Expression);
- Class instance creation;
- Field access;
- Method invocation;
- Array access;
- Array creation.

As stated earlier, a primary expression is broken into *primary components* by the periods in it (except for periods in actual arguments). If there are no periods, the expression has a single primary component. For example, `b.c.getD().foo(x.getY())` is a method invocation expression with 4 components, namely: `b, c, getD()`, and `foo(x.getY())`.

With few exceptions, a primary expression component matches up to a *simple* arc, and a primary expression to an IOM Path. The first component is matched to an initial arc as described below, then the remaining components are matched easily.

3.5.1 Tracing the Initial Component

The Java 1.02 grammar [103] contains the following productions concerning primary expressions:

```
PrimaryExpression :=    PrimaryPrefix ( PrimarySuffix )*
PrimaryPrefix :=      Literal
                |       Name
                |       "this"
                |       "super" "." <IDENTIFIER>
                |       "(" Expression ")"
                |       AllocationExpression
PrimarySuffix :=      "[" Expression "]"
                |       "." <IDENTIFIER>
                |       Arguments
Literal         :=      <INTEGER_LITERAL>
                |       <FLOATING_POINT_LITERAL>
                |       <CHARACTER_LITERAL>
                |       <STRING_LITERAL>
                |       BooleanLiteral
```

```
                |       NullLiteral
BooleanLiteral  :=      "true"
                |       "false"
NullLiteral     :=      "null"
Name            :=      <IDENTIFIER> ( "." <IDENTIFIER> )*
Arguments       :=      "(" ( ArgumentList )? ")"
ArgumentList    :=      Expression ( "," Expression )*
AllocationExpression := "new" PrimitiveType | ArrayDimensions
                |       "new" Name (Arguments | ArrayDimensions)
```

Excluding literals and parenthesized expressions, we need work only with primary expressions which have one of these primary prefixes:

1. Name
2. "this"
3. "super" "." <IDENTIFIER>
4. AllocationExpression

Without too much inconvenience, we can assume that if the prefix is a Name, the first component is either a local variable or parameter, since if it is a field or method name, it can be required to begin with "this". Suppose the primary expression is found within class X. Then the matching to an initial arc in an Extended IOM Graph is as follows:

1. Name - A local arc emanating from X with label matching the first component of the Name. There may be more than one, since variables with the same name may be declared in different scopes. Identifying which is the matching arc requires tracking the environment in which the expression occurs, and this is the approach taken in Chapter 4, for building the prototype. An alternative approach, using *Static Single-Assignment* [36] is mentioned in the section on related work. It involves an initial step which uses scope information to rename variables so that any usage of a variable name can unambiguously be traced back to its definition.
2. "this" - The arc from X to X labelled "this".
3. "super" "." <IDENTIFIER> - The "super" arc emanating from X.
4. AllocationExpression - An expression starting with the word "new" is matched to an Allocation arc with the same label. As with Name, there may be more than one arc emanating from X with the same allocation label. The environment in which the allocation expression occurs determines which one to match.

3.5.2 Tracing the Second and Later Components

The second and later components start with an identifier, since they could only stem from a primary prefix of "super" "." <IDENTIFIER>, the next component of a Name, or a primary suffix of "." <IDENTIFIER>. Having matched the first component to an initial arc, let Y be the target of that arc. If Y is primitive, there can be no more components. Assuming Y is a Ref node, the second component must match a member (field or method) defined for class Y or an ancestor of Y, or else the primary expression would not compile.

Matching a field access is straightforward, since field names are unique. If the field is not available at Y, simply follow "super" arcs from Y, until reaching a node Z which supports the field. Then insert a cast "(Z)" arc, followed by the field arc.

Matching method invocations is a bit complicated. This issue is dealt with at length in the Java Language Specification [45], Section 15.11. It is summarized here.

After determining the name of the method invoked, it is necessary to find method declarations that are *applicable* and *accessible*. If the method invocation contains parameters, their type is determined at compile-time. A method declaration is *applicable* if the invocation and declaration have the same number of parameters, and the type of each actual parameter is either the same as that of the matching formal parameter, or can be widened to it. Note that a reference to a class can be widened to a reference to an ancestor class, an int can be widened to a long, etc. Using $a \leq b$ to mean that b is a widening of a, we have:

- $X \overset{\star}{\Longleftarrow} Y \Longrightarrow X \leq Y$
- $byte \leq short \leq int \leq long \leq float \leq double$;
- $char \leq int$.

For example, the method declaration foo(long, boolean) is applicable to the invocation expression foo(5, false), since the literal 5 has type int, which can be widened to long.

The accessibility of a method declaration is determined partly by the access modifiers (public, none, protected or private). Accessibility then consists of locating method declarations in the current class, or non-private ones in its ancestors, that are applicable. Finally, the most specific applicable, accessible method is chosen to "provide the descriptor for the run-time method dispatch". Informally, a method declaration is more specific than another if it would require less widening of the actual parameters.

Since the final choice of method to be invoked may be deferred until run-time, it is necessary to include usage traces for each of the possible choices. As with fields, when mapping a method invocation from Y, if the chosen method is in an ancestor Z of Y, insert a cast "(Z)" arc, followed by the method arc.

3.6 Relating IOM Graphs, Extended IOM Graphs and Usage Traces

We summarize the results of the preceding two sections.

Start with a Java program P. P contains declarations of classes, fields, methods, and inheritances. These are described by an IOM Graph, and represent constructs amenable to CSL transformations. Now extend P by including declarations from imported classes, as well as local variables and parameters, casts, and allocation expressions. You now have an Extended IOM Graph, Arcs in this graph are either *initial*, or *non-initial*. An *IOM Path* is a path in this graph which starts with an initial arc, and continues with zero or more non-initial arcs.

Now examine the expressions found inside the initializers, and method and constructor bodies of program P. Limit to those composed from casts and non-literal primaries, the so-called *PrimCast(P)* expressions. In Section 3.5, it was shown that any expression found in PrimCast(P) can be traced to a set of IOM paths, provided we know the static scope of any variables in its initial component. We define the map, *trace*:

$$trace : \Gamma \times PrimCast(P) \longrightarrow \{IOMPath\}$$

which maps a PrimCast expression located in environment Γ to a set of IOM paths called its *usage trace*.

In Section 3.4, it was shown that a map:

$$possibleUsages : IOMPath \longrightarrow \{PrimCast\}$$

could be defined to generate possible PrimCast expressions from IOM Paths. We now define the map *usages*:

$$usages : IOMPath \longrightarrow \{PrimCast(P)\}$$

which associates actual expressions to the IOM Paths which generate them. If ip is an IOM Path, then:

$$usages(ip) = \{pc \in PrimCast(P) \mid ip \in trace(pc)\}$$

3.6.1 IOM Path Patterns Using Wild Cards

In the next section, the presence or absence of usages of paths will often be used to determine the safety of proposed transformations. Sets of IOM Paths and PrimCast expressions will sometimes be described by using patterns. An *IOM path pattern* is formed from an IOM Path by replacing some arc labels or path segments by wild cards. Wild cards are asterisks (*), or omitted arc sources or targets. For example, the pattern $\xrightarrow{*}$ X$\xrightarrow{*}$ matches all IOM Paths entering and then leaving node X. Usage mappings can be applied to patterns, thus the statement:

$$usages(\xrightarrow{(X)} X \xrightarrow{foo()}) = \phi$$

is taken to mean that no expression in P consists of a cast to X, (explicit or implicit) followed by an invocation of foo(). The statement:

$$usages(\xrightarrow{*}X\xrightarrow{foo()}) = \phi$$

means there is no usage of the method X::foo().

If W, X, Y are Ref nodes with W$\xleftarrow{\star}$X$\xleftarrow{\star}$Y, we consider that up-casts from W to Y traverse the arc X$\xrightarrow{super}$ in the sense that $W\xrightarrow{(Y)}Y$ is equivalent to:

$$W\xrightarrow{(X)}X\xrightarrow{super} * \xrightarrow{(Y)}Y$$

3.7 Type-Soundness and Behavior Preservation

As mentioned in this chapter's introduction, in considering whether to transform an existing IOM construct, or to add a new one, one has to ask:

1. Will the proposed transformation break the language rules? In particular, will it break the type-soundness of an existing program, preventing its re-compilation?
2. Will it alter the behavior of an program so that, for example, a different output would be obtained from the same input?

We first consider these questions in a general way; later we will consider each of the transformations in the CSL, case by case. The easiest question to deal with is the first. The rules of Java are fairly straightforward, and easily checked by examining the IOM Graph. Here are some of those rules:

1. The inheritance hierarchy must be acyclic.
2. Reserved words may not be used for names.
3. There must be no duplicates among class names.
4. In the same class, there can be no duplicates among field names.
5. In the same class, there cannot be multiple operations with the same signature.
6. A class can extend at most one other class.
7. A cast between reference types must be either an up-cast to an ancestor, or a downcast to a descendent.
8. An operation cannot have a different return type than an inherited one with the same signature.

A harder question is to check for type soundness. The basic issue here is whether the components used in expressions can be resolved to component declarations. This is where the usage traces can be used for checking beforehand. In general it would be unwise to drop a field or operation which had usages within expressions; however if matching fields or operations were inherited from an ancestor class, the expressions could still be resolved and the programs would still compile. But the behavior of the programs might well change, which brings up the third question.

Preservation of behavior after transformation is complicated in object-oriented languages by the fact that some features in ancestor classes are hidden by matching features in descendent classes. Transformations, even when type-sound, may introduce subtle changes by hiding or exposing features. Operations, for example, override based on their signatures, even though their behavior might be quite different. A transformation which resulted in a different method being invoked from an expression would preserve behavior only if the methods were *semantically equivalent*. However, in general, it is not computationally feasible to check for semantic equivalence. The necessary condition for behavior preservation can be defined, but not checked. A second best solution is to define sufficient conditions which would guarantee behavior preservation. That is the approach taken here, however it should be recognized that it is overly restrictive, rejecting some transformations that would actually be safe to make.

Fields hidden by overriding definitions also present a problem. Even though hidden, storage for a field is still provided in an object instance, and the field can in fact be accessed through use of "super", or a cast. Suppose class A extends class B, and each has a field named x of type int. Dropping the field in A would preserve type soundness; however if the x fields in A and B had formerly been used to store different values, this would likely change behavior. In this situation, a careful check of usages should reveal if it is safe to drop.

3.8 Examining the LLP's

This section systematically examines the lower level primitives, including necessary preconditions to apply them safely, and compensations which must be made to expressions which are affected by their application.

Before applying a primitive, the weak precondition, and possibly the strong one also, is checked. Application of the primitive involves modifying certain program declarations, and often also certain expressions found in method bodies and initializations. These modifications result in the postcondition being met. The following template will be used:

- **Primitive:**
- **Example:**
- **Declaration Changes:**
- **Weak Precondition:**
- **Strong Precondition:**

- **PrimCast Changes:**
- **Note:**
- **Postcondition:**

The wording of the primitive indicates the main declaration changes that need to be made such as "ADD HAS-A ...". Hence, the **Declaration Changes** component describes only ancillary declaration changes that may need to be made. Often they can be accomplished using CSL commands.

Preconditions are stated in terms of the Extended IOM Graph, and the usage traces on that graph induced by the PrimCast expressions contained in the program. A transformation which satisfies its weak precondition is guaranteed to preserve the type soundness of the program. **Weak precondition** checking includes locating the constructs which are to be altered by the transformation. In some cases, the necessity of the preconditions is built up from special cases of considering what might go wrong with the transformation. Where the necessary preconditions are not feasible to evaluate, a *sufficient* precondition is stated, which entails the *necessary* ones.

Some of the transformations also specify a **strong precondition**. Here the intent is not only to preserve compilability, but also to preserve the behavior of all methods in the system. For example, adding an operation which overrides an inherited operation could be said to preserve compilability but may drastically alter behavior. Where strong preconditions are not specified, the weak preconditions suffice to preserve behavior.

The **PrimCast Changes** component describes modifications that need to be made to expressions already existing in method bodies and initializations. As discussed earlier, it suffices to limit the description to PrimCast expressions.

Postconditions are stated in terms of the existence or non-existence of constructs in the Extended IOM Graph.

To improve readability, the checking of conditions is sometimes described by referring to the **example** given for the transformation, rather than to its full generality. The following notational conveniences will also be used:

Let P be a program, and let Λ be its Extended IOM Graph. We can then refer to:

- Classes(Λ) = the classes defined in P;
- Imports(Λ) = the classes imported by P;
- Refs(Λ) = Classes(Λ)$\cup$ Imports(Λ);
- Prim(Λ) = the primitive types used in P;
- Nodes(Λ) = Refs(Λ)$\cup$ Prim(Λ);
- Arcs(Λ) - the simple arcs of Λ;

- Paths(Λ) - the IOM Paths of Λ.

Also recall from Section 2.2.2, the notations for class members. Members defined within class C are referred to using lower case as in:

- attributes(C);
- hasas(C);
- fields(C) = $attributes(C) \cup hasas(C)$;
- operations(C);

while uppercase is used to denote members either defined in C or inherited into C as in:

- Attributes(C);
- Hasas(C);
- Fields(C) = $Attributes(C) \cup Hasas(C)$;
- Operations(C).

Furthermore, the *origin* of a member is the class in which the member is defined.

Note on Matching

The term *matching* will apply to fields, operation signatures, and operations with the following sense:

- Fields match if they have the same name and type;
- Operation signatures match if they have the same name, the same number of parameters, and corresponding parameters have the same type;
- Operations match if their signatures match, and they have the same return type;

Examination of the lower level primitives follows:

3.8.1 ADD CLASS

- **Primitive:** ADD CLASS *className*.
- **Example:** ADD CLASS X.
- **Weak Precondition:** X$\notin$ Nodes(Λ).
- **Postcondition:** X$\in$ Classes(Λ).

3.8.2 DROP CLASS

- **Primitive:** DROP CLASS *className*.
- **Example:** DROP CLASS X.
- **Declaration Changes:** If W$\Longleftarrow$X$\Longleftarrow$Y:
  1. DROP IS-A W X;
  2. ADD IS-A W Y.
- **Weak Precondition:**
  1. $X \in$ Classes(Λ);
  2. usages($\xrightarrow{*}$ X $\xrightarrow{*}$) = ϕ; *or*
  3. Every usage of $\xrightarrow{*}$ X $\xrightarrow{*}$ matches the pattern $\xrightarrow{*}$ X $\xrightarrow{super}$ Y$\xrightarrow{*}$, X$\Longleftarrow$Y.
- **PrimCast Changes:** If X is used only as a pass-through for inherited members, (see weak precondition 3) casts to X are replaced by casts to Y, where X$\Longleftarrow$Y.
- **Note:** It is permissable to drop a class which is used only as a pass-through to get to members of an ancestor class.
- **Postcondition:** $X \notin$ Classes(Λ).

3.8.3 RENAME CLASS

- **Primitive:** RENAME CLASS *OldClassName* NEW NAME *NewClassName*.
- **Example:** RENAME CLASS A NEW NAME B.
- **Weak Precondition:**
  1. $A \in$ Classes(Λ)
  2. $B \notin$ Nodes(Λ).
- **PrimCast Changes:** All references to class A in a PrimCast expression are changed to refer to class B.
- **Postcondition:**
  1. $A \notin$ Classes(Λ).
  2. $B \in$ Classes(Λ).

3.8.4 ADD ATTRIBUTE

- **Primitive:** ADD ATTRIBUTE *attributeName* SOURCE *className* TARGET *attributeType*.
- **Example:** ADD ATTRIBUTE attr SOURCE X TARGET T.
- **Weak Precondition:**
  1. X$\in$ Classes(Λ).
  2. T $\in$ Prim.
  3. X$\stackrel{attr}{\longrightarrow}$ * $\notin$ Arcs(Λ). (X doesn't already have a field named attr)
- **PrimCast Changes:** If attr hides an inherited field, say W::attr where X$\stackrel{\star}{\Longleftarrow}$W, then existing PrimCast expressions in X or its descendents which use W::attr must be altered to include an explicit cast, as in (W this).attr.
- **Postcondition:** X$\stackrel{attr}{\longrightarrow}$ T $\in$ Arcs(Λ).

3.8.5 DROP ATTRIBUTE

- **Primitive:** DROP ATTRIBUTE *attributeName* SOURCE *className*,
- **Example:** DROP ATTRIBUTE attr SOURCE X,
- **Weak Precondition:**
  1. $X \in$ Classes(Λ)
  2. X $\stackrel{attr}{\longrightarrow}$ t$\in$ Arcs(Λ), where t $\in$ Prim.
  3. If usages($\stackrel{*}{\longrightarrow}$ X $\stackrel{attr}{\longrightarrow}$) t $\neq \phi$, then X $\stackrel{(Y)}{\longrightarrow}$ Y $\stackrel{attr}{\longrightarrow}$ t $\in$ Paths(Λ), where Y is the first ancestor of X with a field named attr.
- **Strong Precondition:** Either:
  1. usages($\stackrel{*}{\longrightarrow}$ X $\stackrel{attr}{\longrightarrow}$) t = ϕ (no usages of the attribute) or;
  2. X$\stackrel{\star}{\Longleftarrow}$Y, Y $\stackrel{attr}{\longrightarrow}$ t $\in$ Arcs(Λ), where Y is the first ancestor of X with a field named attr, and usages(Z $\stackrel{(Y)}{\longrightarrow}$ Y $\stackrel{attr}{\longrightarrow}$) = ϕ for any Z, Z $\stackrel{\star}{\Longleftarrow}$X (No usages in this class or its descendents of the hidden field supplied from Y)
- **Note:** If dropping X::attr unhides an inherited field Y::attr of the same type, and if there is not already a usage of Y::attr from X or a descendent (say by a cast), then programs will work as before.
- **Postcondition:** X $\stackrel{attr}{\longrightarrow}$ *$\notin$ Arcs(Λ).

3.8.6 RENAME ATTRIBUTE

- **Primitive:** RENAME ATTRIBUTE *oldAttributeName* SOURCE *className* NEW NAME *new attributeName*.
- **Example:** RENAME ATTRIBUTE a1 SOURCE X NEW NAME a2.
- **Weak Precondition:**
  1. X$\in$ Classes(Λ)
  2. X $\xrightarrow{a1}$ t$\in$ Arcs(Λ), where t$\in$ Prim.
  3. X $\xrightarrow{a2}$ *$\notin$ Arcs(Λ).
- **PrimCast Changes:**
  1. A usage of X::a1 from X - rename component a1 to a2.
  2. A usage of X::a1 from a descendent Y of X - Replace component a1 by (X).a2. (Making the cast explicit avoids the problem that field A::a2 may be overridden in Y).
  3. A usage of W::a2 where X$\overset{\star}{\Longleftarrow}$W, from X or a descendent of X - Replace component a2 by (W).a2. (since W::a2 will be overridden by X::a2).
- **Note:** Renaming a field from a1 to a2 may unhide an inherited field a1 and hide an inherited field a2. This is compensated for by including explicit casts (see the PrimCast changes).
- **Postcondition:**
  1. X $\xrightarrow{a1}$ t$\notin$ Arcs(Λ).
  2. X $\xrightarrow{a2}$ t$\in$ Arcs(Λ).

3.8.7 RETARGET ATTRIBUTE

- **Primitive:** RETARGET ATTRIBUTE *attributeName* SOURCE *className* NEW TARGET *newAttributeType*.
- **Example:** RETARGET ATTRIBUTE attr SOURCE X NEW TARGET T.
- **Weak Precondition:**
  1. $X \in$ Classes(Λ)
  2. X $\xrightarrow{attr}$ P$\in$ Arcs(Λ), where P$\in$ Prim.
  3. T $\in$ Prim.
  4. If usages($\xrightarrow{*}$ X $\xrightarrow{attr}$ P) $\neq \phi$, then T is a widening of P. For example, double widens float.

- **PrimCast Changes:** When the new type, t2 of attr represents a widening of the old type t1, then a downcast to t1 must be prefixed to any usage of attr as an r-value, that is as an actual parameter, or the on the right hand side of an assignment.
- **Postcondition:**X $\xrightarrow{attr}$ T$\in$ Arcs(Λ).

3.8.8 REMULTIPLY ATTRIBUTE

- **Primitive:** REMULTIPLY ATTRIBUTE *attributeName* SOURCE *className* NEW MULTIPLICITY (0 | 1).
- **Example:** REMULTIPLY ATTRIBUTE attr SOURCE X NEW MULTIPLICITY m (where m = 0 | 1).
- **Weak Precondition:**
  1. $X \in$ Classes(Λ)
  2. X $\xrightarrow{attr}$ P$\in$ Arcs(Λ), where P$\in$ Prim.
  3. If usages($\xrightarrow{*}$ X $\xrightarrow{attr}$ P) $\neq \phi$, then m = 0
- **PrimCast Changes:** If attr changes from multiplicity 1 to multiplicity 0 (many), existing usages of attr are made to access attr[0]. (Changes from 0 to 1 are not allowed if there are existing usages).
- **Postcondition:**
  1. If m = 0, then X $\xrightarrow{attr[]}$ P $\in$ Arcs(Λ).
  2. If m = 1, then X $\xrightarrow{attr}$ P $\in$ Arcs(Λ).

3.8.9 ADD HAS-A

- **Primitive:** ADD HAS-A *has-aName* SOURCE *sourceClassName* TARGET *Target-ClassName* MULTIPLICITY (0 | 1) [INVERSE *inverseHas-aName*]
- **Example:**
  1. ADD HAS-A ha SOURCE X TARGET Z MULTIPLICITY m. (where m = 0 | 1)
  2. ADD HAS-A ha SOURCE X TARGET Z MULTIPLICITY m INVERSE h2. (using the optional inverse)
- **Weak Precondition:**
  1. X$\in$ Classes(Λ);
  2. Y$\in$ Refs(Λ);
  3. X$\xrightarrow{ha}$ * $\notin$ Arcs(Λ). (X doesn't already have a field named ha)

- **PrimCast Changes:** If ha hides an inherited field, say W::ha where X$\stackrel{\star}{\Longleftarrow}$W, then existing PrimCast expressions in X or its descendents which use W::ha must be altered to include an explicit cast, as in (W this).ha.

- **Postcondition:** X$\stackrel{ha}{\longrightarrow}$ Z $\in$ Arcs(Λ)

3.8.10 DROP HAS-A

- **Primitive:** DROP HAS-A *has-aName* SOURCE *className*.

- **Example:** DROP HAS-A ha SOURCE X.

- **Declaration Changes:** If ha, X$\stackrel{ha}{\longrightarrow}$ T has an optional inverse link, T$\stackrel{haInv}{\longrightarrow}$ X, then haInv is adjusted to drop its inverse.

- **Weak Precondition:**

  1. $X \in$ Classes(Λ)
  2. X $\stackrel{ha}{\longrightarrow}$ T$\in$ Arcs(Λ), where $T \in$ Refs(Λ).
  3. If usages($\stackrel{*}{\longrightarrow}$ X $\stackrel{ha}{\longrightarrow}$ T) $\neq \phi$, then X $\stackrel{(Y)}{\longrightarrow}$ Y $\stackrel{ha}{\longrightarrow}$ T $\in$ Paths(Λ), where Y is the first ancestor of X with a field named ha.

- **Strong Precondition:** Either:

  1. usages($\stackrel{*}{\longrightarrow}$ X $\stackrel{ha}{\longrightarrow}$ T) $= \phi$ (no usages of the HAS-A) or;
  2. X$\stackrel{\star}{\Longleftarrow}$Y, Y $\stackrel{ha}{\longrightarrow}$ T $\in$ Arcs(Λ), where Y is the first ancestor of X with a field named ha, and usages(Z $\stackrel{(Y)}{\longrightarrow}$ Y $\stackrel{ha}{\longrightarrow}$ T) $= \phi$ for any Z, Z $\stackrel{\star}{\Longleftarrow}$X (No usages in this class or its descendents of the hidden field supplied from Y)

- **Note:** If dropping X::ha unhides an inherited field Y::ha of the same type, and if there is not already a usage of Y::ha from X or a descendent (say by a cast), then programs will work as before.

- **Postcondition:** X $\stackrel{ha}{\longrightarrow}$ * $\notin$ Arcs(Λ).

3.8.11 RENAME HAS-A

- **Primitive:** RENAME HAS-A *oldHas-aName* SOURCE *sourceClassName* NEW NAME *newHas-aName*.

- **Example:** RENAME HAS-A h1 SOURCE X NEW NAME h2.

- **Declaration Changes:** If X$\stackrel{h1}{\longrightarrow}$ T is the inverse of T$\stackrel{g}{\longrightarrow}$ X, then the inverse of g is renamed to h2.

- **Weak Precondition:**

1. $X \in$ Classes(Λ)
2. X $\xrightarrow{h1}$ T$\in$ Arcs(Λ), where $T \in$ Refs(Λ).
3. X $\xrightarrow{h2}$ *$\notin$ Arcs(Λ).

- **PrimCast Changes:**
  1. A usage of X::h1 from X - rename component h1 to h2.
  2. A usage of X::h1 from a descendent Y of X - Replace component h1 by (X).h2. Making the cast explicit avoids the problem that field A::h2 may be overridden in Y.
  3. A usage of W::h2 where X$\overset{\star}{\Longleftarrow}$W, from X or a descendent of X - Replace component h2 by (W).h2. (since W::h2 will be overridden by X::h2).
- **Note:** Renaming a field from h1 to h2 may unhide an inherited field h1 and hide an inherited field h2. This is compensated for by making use of casts (see the PrimCast changes). A HAS-A can have an inverse HAS-A which would need to rename its inverse.
- **Postcondition:**
  1. X $\xrightarrow{h1}$ T$\notin$ Arcs(Λ).
  2. X $\xrightarrow{h2}$ T$\in$ Arcs(Λ).

3.8.12 REMULTIPLY HAS-A

- **Primitive:** REMULTIPLY HAS-A *has-aName* SOURCE *sourceClassName* NEW MULTIPLICITY (0 | 1).
- **Example:** REMULTIPLY HAS-A ha SOURCE X NEW MULTIPLICITY m. (where m = 0 | 1)
- **Weak Precondition:**
  1. $X \in$ Classes(Λ);
  2. X $\xrightarrow{ha}$ T$\in$ Arcs(Λ), where T$\in$ Refs(Λ);
  3. If usages($\xrightarrow{*}$ X $\xrightarrow{ha}$) $\neq \phi$, then m = 0.
- **PrimCast Changes:** If ha changes from multiplicity 1 to multiplicity 0 (many), existing usages of ha are made to access ha[0]. (Changes from 0 to 1 are not allowed if there are existing usages).
- **Postcondition:**
  1. If m = 0, then X $\xrightarrow{ha[]}$ T $\in$ Arcs(Λ).
  2. If m = 1, then X $\xrightarrow{ha}$ T $\in$ Arcs(Λ).

3.8.13 RETARGET HAS-A

- **Primitive:** RETARGET HAS-A *has-aName* SOURCE *sourceClassName* NEW TARGET *NewTargetClassName*.
- **Example:** RETARGET HAS-A ha SOURCE X NEW TARGET A.
- **Declaration Changes:** If ha, X$\xrightarrow{ha}$ B had an optional inverse link relationship with a HAS-A haInv, B$\xrightarrow{haInv}$ X, the link is dropped from both after retargeting.
- **Weak Precondition:**
  1. $X \in$ Classes(Λ);
  2. $A \in$ Refs(Λ);
  3. X $\xrightarrow{ha}$ B$\in$ Arcs(Λ), where $B \in$ Refs(Λ);
  4. If usages($\xrightarrow{*}$ X $\xrightarrow{ha}$ B) $\neq \phi$, then B$\xLeftarrow{\star}$A.
- **PrimCast Changes:** When the new target, A of ha represents a widening of the old target B, then a downcast to B must be prefixed to any usage of ha as an r-value, that is, as an actual parameter, or the RHS of an assignment. (The cast will succeed for existing usages, since they were always expressions of type B).
- **Postcondition:** X $\xrightarrow{ha}$ A$\in$ Arcs(Λ).

3.8.14 ADD OPERATION

- **Primitive:** ADD OPERATION *operationName* SOURCE *className* PARAMETERS (*type1*, *type2*, ...) TARGET *typeName*.
- **Example:** ADD OPERATION foo SOURCE X PARAMETERS (T1, T2) TARGET T. (where T1, T2 $\in$ Nodes(Λ))
- **Weak Precondition:**
  1. $X \in$ Classes(Λ);
  2. T1, T2, T $\in$ Nodes(Λ);
  3. X$\xrightarrow{foo(T1,T2)}$ *$\notin$ Arcs(Λ);
  4. If X$\xLeftarrow{\star}$Y and Y$\xrightarrow{foo(T1,T2)}$ U$\in$ Arcs(Λ), then U = T.
- **Strong Precondition:** If X$\xLeftarrow{\star}$Y, then Y$\xrightarrow{foo(T1,T2)} \notin$ Arcs(Λ).
- **Note:** In Java, a redefinition of an operation in a subclass is not allowed to overwrite its return type, hence the second weak precondition. However, the redefinition may well override the behavior, both in terms on the value returned for a given input, and the side effects on the environment, hence a strong precondition is provided.
- **Postcondition:** X$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ).

3.8.15 DROP OPERATION

- **Primitive:** DROP OPERATION *operationName* SOURCE *className* PARAMETERS (*type1*, *type2*, ...).
- **Example:** DROP OPERATION op SOURCE X PARAMETERS (T1,T2).
- **Weak Precondition:**
  1. $X \overset{op(T1,T2)}{\longrightarrow} T \in Arcs(\Lambda)$ for some $T \in Nodes(\Lambda)$
  2. If $usages(X \overset{op(T1,T2)}{\longrightarrow} T) \neq \phi$, then $X \overset{\star}{\Longleftarrow} Y$ and $Y \overset{op(T1,T2)}{\longrightarrow} T \in Arcs(\Lambda)$.
- **Strong Precondition:** $usages(X \overset{op(T1,T2)}{\longrightarrow} T) = \phi$.
- **Note:** An inherited operation may not have the same behavior and side effects, hence the strong precondition.
- **Postcondition:** $X \overset{op(T1,T2)}{\longrightarrow} * \notin Paths(\Lambda)$.

3.8.16 RENAME OPERATION

- **Primitive:** RENAME OPERATION *oldOperationName* SOURCE *className* PARAMETERS (*type1*, *type2*, ..) NEW NAME *newOperationName*.
- **Example:** RENAME OPERATION op1 SOURCE X PARAMETERS (T1,T2) NEW NAME op2.
- **Weak Precondition:**
  1. $X \overset{op1(T1,T2)}{\longrightarrow} T \in Arcs(\Lambda)$;
  2. $X \overset{op2(T1,T2)}{\longrightarrow} * \notin Arcs(\Lambda)$;
  3. If $X \overset{\star}{\Longleftarrow} Y$ and $Y \overset{op2(T1,T2)}{\longrightarrow} U \in Arcs(\Lambda)$, then $U = T$.
- **Strong Precondition:** If $W \overset{\star}{\Longleftarrow} X \overset{\star}{\Longleftarrow} Y$, and $Y \overset{op2(T1,T2)}{\longrightarrow} U \in Arcs(\Lambda)$, then $usages(W \overset{(Y)}{\longrightarrow} Y \overset{op2(T1,T2)}{\longrightarrow}) \; U = \phi$.
- **Note:** The strong precondition is needed, since there is no way to redirect usages of $Y \overset{op2(T1,T2)}{\longrightarrow} U$ so that they will not use the overriding operation $X \overset{op2(T1,T2)}{\longrightarrow} T$.
- **PrimCast Changes:** Usages of $X \overset{op1(T1,T2)}{\longrightarrow} T$ are changed to use $X \overset{op2(T1,T2)}{\longrightarrow} T$.
- **Postcondition:**
  1. $X \overset{op1(T1,T2)}{\longrightarrow} * \notin Paths(\Lambda)$;
  2. $X \overset{op2(T1,T2)}{\longrightarrow} T \in Paths(\Lambda)$;

3.8.17 RETARGET OPERATION

- **Primitive:** RETARGET OPERATION *operationName* SOURCE *className* PARAMETERS (*type1*, *type2*, ...) NEW TARGET *newType*.
- **Example:** RETARGET OPERATION op SOURCE X PARAMETERS (T1,T2) NEW TARGET A.
- **Weak Precondition:**
  1. $X \xrightarrow{op(T1,T2)} B \in Arcs(\Lambda)$ for some $B \in Nodes(\Lambda)$;
  2. $A \in Nodes(\Lambda)$;
  3. If $usages(X \xrightarrow{op(T1,T2)} B) \neq \phi$, then $B \overset{\star}{\Longleftarrow} A$.
- **PrimCast Changes:** If A is a widening of the old target B, existing usages of X::op(T1,T2) are altered to insert a downcast to B of the operation result. (The downcast is valid since X::op(T1,T2) currently returns type B results).
- **Postcondition:** $X \xrightarrow{op(T1,T2)} A \in Arcs(\Lambda)$.

3.8.18 ADD IS-A

- **Primitive:** ADD IS-A *subClassName superClassName*.
- **Example:** ADD IS-A X Y.
- **Weak Precondition:**
  1. $X \in Classes(\Lambda)$;
  2. $Y \in Refs(\Lambda)$;
  3. If X and Y each define or inherit a matching operation, then they must have the same return types.
  4. If $X \xrightarrow{super} Z \in Arcs(\Lambda)$ for some $Z \in Refs(\Lambda)$, then $Y \overset{\star}{\Longleftarrow} Z$.
- **Strong Precondition:** $X \xrightarrow{super} * \notin Arcs(\Lambda)$.
- **PrimCast Changes:** Any usage of an inherited field from an X object: $\xrightarrow{*} X \xrightarrow{super} Z \xrightarrow{f}$, is changed to include an explicit cast to Z.
- **Note:** The weak precondition makes sure that X objects maintain access to any inherited members, but does not guarantee that the behavior of inherited operations is unchanged, hence the strong precondition.
- **Postcondition:** $X \xrightarrow{super} Y \in Arcs(\Lambda)$.

3.8.19 DROP IS-A

- **Primitive:** DROP IS-A *subClassName superClassName.*
- **Example:** DROP IS-A X Y.
- **Weak Precondition:**
  1. $X \xrightarrow{super} Y \in Arcs(\Lambda)$;
  2. $usages(W \xrightarrow{(Y)} Y \xrightarrow{*} *) = \phi$ for any $W \xleftarrow{\star} X$;
  3. $usages(X \xrightarrow{super} Y \xrightarrow{*} *) = \phi$.
- **Note:** Dropping the IS-A $X \Longleftarrow Y$ is generally harmful since it reduces the properties available to X, however if X already overrides them there is no effect. This would be the case, for example, if DROP IS-A X Y was part of the expansion of the higher level primitive, FLATTEN Y.
- **Postcondition:** $X \xrightarrow{super} Y \notin Arcs(\Lambda)$.

3.8.20 DELEGATE FIELD

- **Primitives:**
  1. DELEGATE FIELD *fieldName* SOURCE *sourceClassName* USING HAS-A *hasaName* .
  2. DELEGATE FIELD *fieldName* SOURCE *sourceClassName* USING OPERATION *operationName.*
- **Examples:**
  1. DELEGATE FIELD f SOURCE X USING HAS-A y.
  2. DELEGATE FIELD f SOURCE X USING OPERATION getY.
- **Weak Precondition:**
  1. $X \xrightarrow{f} S \in Arcs(\Lambda)$ for some $S \in Nodes(\Lambda$;
  2. Either $X \xrightarrow{y} Y \in Arcs(\Lambda)$ or $X \xrightarrow{getY()} Y \in Arcs(\Lambda)$ for some $Y \in Classes(\Lambda)$;
  3. If $Y \xrightarrow{f} T \in Arcs(\Lambda)$ for some $T \in Nodes(\Lambda)$, then $T = S$.
- **Strong Precondition:** $Y \xrightarrow{f} * \notin Arcs(\Lambda)$.
- **PrimCast Changes:** PrimCast expressions which use the arc $X \xrightarrow{f}$, are altered to include either a "y." or "getY()." before the "f".

- **Note:** DELEGATE FIELD is included to support the higher level primitive, AGGREGATE, which may be used to delegate fields from several classes at once. The strong precondition is checked only when field f from the first such class is delegated. If Y already has a field named f, the transformation is rejected. When field f from subsequent classes is delegated, the strong precondition is not checked, allowing sharing of Y::f.
- **Postcondition:**
  1. $X \xrightarrow{f} * \notin \text{Arcs}(\Lambda)$;
  2. $Y \xrightarrow{f} S \in \text{Arcs}(\Lambda)$.

3.8.21 RECLAIM FIELD

- **Primitives:**
  1. RECLAIM FIELD *fieldName* SOURCE *sourceClassName* USING HAS-A *hasaName*.
  2. RECLAIM FIELD *fieldName* SOURCE *sourceClassName* USING OPERATION *operationName*.
- **Examples:**
  1. RECLAIM FIELD f SOURCE X USING HAS-A y.
  2. RECLAIM FIELD f SOURCE X USING OPERATION getY.
- **Declaration Changes:** Add field arc $X \xrightarrow{f} S$, where S is the type of Y::f, if it does not already exist.
- **Weak Precondition:**
  1. Either $X \xrightarrow{y} Y \xrightarrow{f} S \in \text{Arcs}(\Lambda)$ or $X \xrightarrow{getY()} Y \xrightarrow{f} S \in \text{Arcs}(\Lambda)$ for some $S \in \text{Nodes}(\Lambda$;
  2. If $X \xrightarrow{f} T \in \text{Arcs}(\Lambda)$ for some $T \in \text{Nodes}(\Lambda$, then $T = S$.
- **Strong Precondition:** $X \xrightarrow{f} * \notin \text{Arcs}(\Lambda)$.
- **PrimCast Changes:**
  1. PrimCast expressions which use the arc $X \xrightarrow{y} Y \xrightarrow{f} S$ or $X \xrightarrow{getY()} Y \xrightarrow{f} S$, are altered to remove either the "y." or "getY()." before the "f".
  2. If arc $X \xrightarrow{f} S$ needs to be added, and it hides an inherited field $W \xrightarrow{f} U$ where $X \xleftarrow{+} W$, then usages of $W \xrightarrow{f} U$ from X or a descendent, are altered to include an explicit cast to W as in $\xrightarrow{(W)} W \xrightarrow{f} U$.
- **Note:** The strong precondition protects any usages of an existing arc $X \xrightarrow{f}$. These cannot be compensated for by casting.
- **Postcondition:** $X \xrightarrow{f} S \in \text{Arcs}(\Lambda)$.

3.8.22 DELEGATE OPERATION

- **Primitives:**
  1. DELEGATE OPERATION *operationName* SOURCE *sourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*.
  2. DELEGATE OPERATION *operationName* SOURCE *sourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*.
- **Examples:**
  1. DELEGATE OPERATION foo SOURCE X PARAMETERS (T1,T2) USING HAS-A y.
  2. DELEGATE OPERATION foo SOURCE X PARAMETERS (T1,T2) USING OPERATION getY.
- **Declaration Changes:**
  1. If $Y \overset{foo(T1,T2)}{\longrightarrow} * \notin \text{Arcs}(\Lambda)$, copy X::foo(T1,T2) to Y, including the method body.
  2. Drop $X \overset{foo(T1,T2)}{\longrightarrow} S$.
- **Weak Precondition:**
  1. $X \overset{foo(T1,T2)}{\longrightarrow} S \in \text{Arcs}(\Lambda)$ for some $S \in \text{Nodes}(\Lambda)$;
  2. Either $X \overset{y}{\longrightarrow} Y \in \text{Arcs}(\Lambda)$ or $X \overset{getY()}{\longrightarrow} Y \in \text{Arcs}(\Lambda)$ for some $Y \in \text{Classes}(\Lambda)$.
  3. If $W \overset{foo(T1,T2)}{\longrightarrow} T \in \text{Arcs}(\Lambda)$ for some W such that $Y \overset{\star}{\Longleftarrow} W$ and some $T \in \text{Nodes}(\Lambda)$, then T = S.
  4. If there is any usage inside the method body of X::foo(T1,T2) of a field $f \in \text{Fields}(X)$, then Fields(Y) also includes a field f, and it has the same type.
  5. If there is any usage inside the method body of X::foo(T1,T2) of an operation bah(...) $\in$ Operations(X), then Operations(Y) contains a signature matching operation bah(...), and it has the same return type.
- **Strong Precondition:**
  1. $W \overset{foo(T1,T2)}{\longrightarrow} \notin \text{Arcs}(\Lambda)$ for any W such that $Y \overset{\star}{\Longleftarrow} W$.
  2. If there is any usage inside the method body of X::foo(T1,T2) of a field $f \in \text{Fields}(X)$, then the matching field f in Fields(Y) required by the weak precondition, has the same origin.
  3. If there is any usage inside the method body of X::foo(T1,T2) of an operation bah(...) $\in$ Operations(X), then at least one of the following is true:
     (a) bah is foo(T1,T2); or
     (b) bah is invoked using the x.bah(...) syntax; or

(c) bah is invoked using the unqualified bah(...) syntax, but the bah(...) $\in$ Operations(X) has the same origin as that in Operations(Y), and furthermore does not reference any field, or invoke any method that is overridden in X or Y.

- **PrimCast Changes:** PrimCast expressions which use the arc $X \stackrel{foo(T1,T2)}{\longrightarrow}$, are altered to include either a "y." or "getY()." before the "foo(T1,T2)".
- **Note:** DELEGATE OPERATION is included to support the higher level primitive, AGGREGATE, which may be used to delegate operations from several classes at once. The strong condition for DELEGATE OPERATION is so restrictive that it can only rarely be satisfied. On the other hand, even when AGGREGATE is exercised using strong checking, this may be safely relaxed for the DELEGATE OPERATION's which are spawned.
- **Postcondition:**
  1. $X \stackrel{foo(T1,T2)}{\longrightarrow} S \notin \text{Arcs}(\Lambda)$;
  2. $Y \stackrel{foo(T1,T2)}{\longrightarrow} S \in \text{Arcs}(\Lambda)$.

3.8.23 RECLAIM OPERATION

- **Primitives:**
  1. RECLAIM OPERATION *operationName* SOURCE *sourceClassName* PARAMETERS (*type1*, *type2*, ...) USING HAS-A *has-aName*.
  2. RECLAIM OPERATION *operationName* SOURCE *sourceClassName* PARAMETERS (*type1*, *type2*, ...) USING OPERATION *operationName*.
- **Examples:**
  1. RECLAIM OPERATION foo SOURCE X PARAMETERS (T1,T2) USING HAS-A y.
  2. RECLAIM OPERATION foo SOURCE X PARAMETERS (T1,T2) USING OPERATION getY().
- **Declaration Changes:** Arc $X \stackrel{foo(T1,T2)}{\longrightarrow} S$ is added, provided there is not already an arc $W \stackrel{foo(T1,T2)}{\longrightarrow} *$ for some W, $X \stackrel{\star}{\Longleftarrow} W$.
- **Weak Precondition:**
  1. Either $X \stackrel{y}{\longrightarrow} Y \stackrel{foo(T1,T2)}{\longrightarrow} S \in \text{Arcs}(\Lambda)$ or $X \stackrel{getY()}{\longrightarrow} Y \stackrel{foo(T1,T2)}{\longrightarrow} S \in \text{Arcs}(\Lambda)$ for some $S \in \text{Nodes}(\Lambda)$;
  2. If $W \stackrel{foo(T1,T2)}{\longrightarrow} T \in \text{Arcs}(\Lambda)$ for some W, $X \stackrel{\star}{\Longleftarrow} W$ and some $T \in \text{Nodes}(\Lambda)$, then $T = S$.
  3. If there is any usage inside the method body of Y::foo(T1,T2) of a field $f \in$ Fields(Y), then Fields(X) also includes a field f, and it has the same type.

4. If there is any usage inside the method body of Y::foo(T1,T2) of an operation bah(...) $\in$ Operations(Y), then Operations(X) contains a signature matching operation bah(...), and it has the same return type.

- **Strong Precondition:**

  1. $W \xrightarrow{foo(T1,T2)} * \notin Arcs(\Lambda)$ for any W such that $X \overset{\star}{\Longleftarrow} W$.
  2. If there is any usage inside the method body of Y::foo(T1,T2) of a field f $\in$ Fields(Y), then the matching field f in Fields(X) required by the weak precondition, has the same origin.
  3. If there is any usage inside the method body of Y::foo(T1,T2) of an operation bah(...) $\in$ Operations(X), then at least one of the following is true:
     (a) bah is foo(T1,T2); or
     (b) bah is invoked using the x.bah(...) syntax; or
     (c) bah is invoked using the unqualified bah(...) syntax, but the bah(...) $\in$ Operations(X) has the same origin as that in Operations(Y), and furthermore does not reference any field, or invoke any method that is overridden in X or Y.

- **PrimCast Changes:** PrimCast expressions which use the paths $X \xrightarrow{y} Y \xrightarrow{foo(T1,T2)}$ or $X \xrightarrow{getY()} Y \xrightarrow{foo(T1,T2)}$, are altered to remove either the "y." or "getY()." before the "foo(T1,T2)".

- **Note:** RECLAIM OPERATION is used to support the higher level primitive, DECOMPOSE. The strong precondition is necessary to ensure that if operation foo(T1,T2) was already available at class X, its behavior is not overridden.

- **Postcondition:** $X \xrightarrow{foo(T1,T2)} S \in Arcs(\Lambda)$.

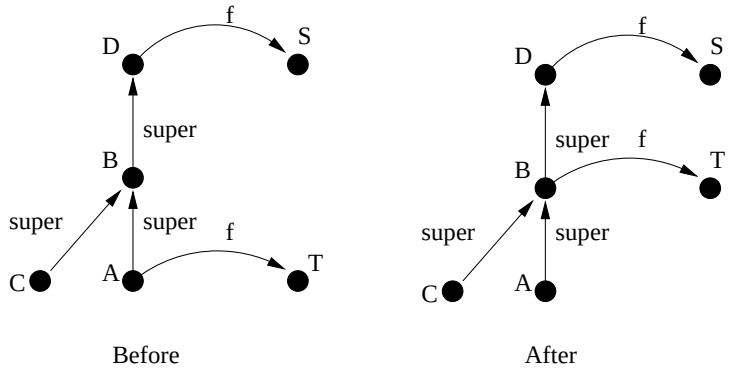

Figure 3.7: After field f is lifted from A to B, a usage of D::f from C requires an explicit cast, as in ((D) c).f, where c$\in$ C.

3.8.24 LIFT FIELD

- **Primitive:** LIFT FIELD *FieldName* SOURCE *className1* TO *className2*.
- **Example:** LIFT FIELD f SOURCE A TO B.
- **Weak Precondition:**
  1. A$\xrightarrow{f}$ T $\in$ Arcs(Λ) for some T $\in$ Nodes(Λ);
  2. A$\Longleftarrow$B;
  3. If B$\xrightarrow{f}$ S $\in$ Arcs(Λ) for some S $\in$ Nodes(Λ), S = T.
- **Strong Precondition:** B$\xrightarrow{f}$ *$\notin$ Arcs(Λ).
- **PrimCast Changes:** If there is any usage of a path: B$\xrightarrow{super}$ * $\xrightarrow{*}$ D$\xrightarrow{f}$, it is altered by inserting an explicit cast to D before accessing field f. (See Figure 3.7).
- **Note:** LIFT FIELD is used to support FACTOR, which lifts common members of several input classes into a joint superclass. The strong precondition is checked only when a field from the first input class is lifted. The field is then created in the superclass, and fields from the other input classes are lifted.
- **Postcondition:**
  1. A$\xrightarrow{f}$ * $\notin$ Arcs(Λ);
  2. B$\xrightarrow{f}$ T $\in$ Arcs(Λ).

3.8.25 LIFT OPERATION

- **Primitive:** LIFT OPERATION *operationName* SOURCE *className1* PARAMETERS (*type1*, *type2*, ...) TO *className2*.
- **Example:** LIFT OPERATION foo SOURCE A PARAMETERS (T1,T2) TO B.
- **Weak Precondition:**
  1. A$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ) for some T $\in$ Nodes(Λ);
  2. A$\Longleftarrow$B;
  3. If there is any usage inside the method body of A::foo(T1,T2) of a field f $\in$ Fields(A), then Fields(B) also includes a field f, and it has the same type.
  4. If there is any usage inside the method body of A::foo(T1,T2) of an operation bah(...) $\in$ Operations(A), then Operations(B) contains a signature matching operation bah(...), and it has the same return type.
- **Strong Precondition:**
  1. W$\xrightarrow{foo(T1,T2)}$ S $\notin$ Arcs(Λ), for any W such that B$\overset{\star}{\Longleftarrow}$W.

2. If there is any usage inside the method body of B::foo(T1,T2) of a field f $\in$ Fields(B), f is not hidden in A by a field A$\xrightarrow{f}$ *.
3. If there is any usage inside the method body of A::foo(T1,T2) of an operation A$\xrightarrow{bah(...)}$ S, then at least one of the following is true:
   (a) bah is foo(T1,T2); or
   (b) bah is invoked using the x.bah(...) syntax; or
   (c) bah is invoked using the unqualified bah(...) syntax, but bah is not overridden in A.

- **Postcondition:**
  1. A$\xrightarrow{foo(T1,T2)}$ * $\notin$ Arcs(Λ);
  2. B$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ).

3.8.26 LOWER FIELD

- **Primitive:** LOWER FIELD *FieldName* SOURCE *className1* TO *className2*.
- **Example:** LOWER FIELD f SOURCE B TO A.
- **Weak Precondition:**
  1. A$\Longleftarrow$B;
  2. B$\xrightarrow{f}$ T $\in$ Arcs(Λ) for some T $\in$ Nodes(Λ);
  3. If A$\xrightarrow{f}$ T' $\in$ Arcs(Λ), then T' = T.
- **Strong Precondition:** A$\xrightarrow{f}$ * $\notin$ Arcs(Λ).
- **Postcondition:** A$\xrightarrow{f}$ T $\in$ Arcs(Λ).
- **PrimCast Changes:** Any usage of B::f from A that contained an explicit cast to A, should have the cast removed.
- **Note:** With weak condition checking, override is permitted provided the field types match. With strong checking, LOWER FIELD is rejected if it requires an override.

3.8.27 LOWER OPERATION

- **Primitive:** LOWER OPERATION *operationName* SOURCE *className1* PARAMETERS (*type1*, *type2*, ...) TO *className2*.
- **Example:** LOWER OPERATION foo SOURCE B PARAMETERS (T1,T2) TO A.
- **Weak Precondition:**
  1. A$\Longleftarrow$B.

2. B $\stackrel{foo(T1,T2)}{\longrightarrow}$ T $\in$ Arcs(Λ) for some T $\in$ Nodes(Λ);

- **Strong Precondition:**
  1. A $\stackrel{foo(T1,T2)}{\longrightarrow}$ * $\notin$ Arcs(Λ).
  2. If the method body of B::foo(T1,T2) invokes a method bah at least one of the following is true:
     (a) bah is foo(T1,T2); or
     (b) bah is invoked using the x.bah(...) syntax; or
     (c) bah is invoked using the unqualified bah(...) syntax, but bah is not overridden in A.
- **Postcondition:** A $\stackrel{foo(T1,T2)}{\longrightarrow}$ T $\in$ Arcs(Λ).
- **PrimCast Changes:** If there is any usage inside the method body of B::foo(T1,T2) of a field f which is overridden in A, then after lowering, an explicit cast to B is inserted before the usage.
- **Note:** When weak condition checking is indicated, LOWER OPERATION will override a matching operation in the subclass, provided the return types match. When strong condition checking is indicated, LOWER OPERATION would fail if it meant an override.

3.9 Examining the HLP's

When a higher level primitive is exercised, it spawns a number of potential lower level primitives. Checking the preconditions of these LLP's will determine which ones will actually be carried out. For example, the Factor HLP defined below spawns LLP primitives LIFT FIELD and LIFT OPERATION for each field or operation respectively which is common to its input classes. Any of these LLP's which violate their precondition would be rejected.

Consider the command: FACTOR A, B INTO C, where A and B each have an attribute named f. It spawns:

1. ADD CLASS C
2. ADD IS-A A C
3. ADD IS-A B C
4. LIFT FIELD f SOURCE A TO C
5. LIFT FIELD f SOURCE B TO C

FACTOR requires a kind of atomicity in that either both of the spawned LIFT FIELD f commands must succeed or neither succeeds. This is indicated by enclosing these commands inside a transaction as follows:

1. ADD CLASS C
2. ADD IS-A A C
3. ADD IS-A B C

   BEGIN TRANSACTION

4. LIFT FIELD f SOURCE A TO C
5. LIFT FIELD f SOURCE B TO C†

   END TRANSACTION

Here the † indicates that the spawned primitive is to be executed under weak precondition checking, even when FACTOR is executed under strong precondition checking. The reason is that strong checking would reject LIFT FIELD f SOURCE B TO C, since C would already have a field f from A.

A higher level primitive should be regarded as a kind of macro. It is expanded into a sequence of LLP's or other HLP's, the exact nature of which depends on the current state of the IOM. In the discussion of the individual HLP's, this expansion will be shown given a sample IOM, which is sufficient to bring out the nature of the expansion.

The reader will be interested in knowing what declaration changes to the sample IOM are guaranteed when the preconditions of the HLP are met. The situation is analagous to a forward-chaining production system [99]. Initial facts are entered into working memory from the state of the sample IOM and the preconditions of the HLP. The spawned primitives are considered one-at-a-time in order. If the facts in working memory satisfy a spawned primitive's preconditions, it is fired, and working memory is updated according to its postconditions. After all spawned preconditions have been considered, the set of facts in working memory, less any that were in the initial set, is presented as the postcondition of the HLP.

3.9.1 FACTOR

- **Primitive:** FACTOR $className_1$, $className_2$, ... INTO *abstractClassName*.
- **Example:** FACTOR A, B INTO C, assuming:

  A1. $A \xrightarrow{f} T \in \text{Arcs}(\Lambda)$

  A2. $B \xrightarrow{f} T \in \text{Arcs}(\Lambda)$

  A3. $A \overset{foo(T1,T2)}{\longrightarrow} T \in \text{Arcs}(\Lambda)$

  A4. $B \overset{foo(T1,T2)}{\longrightarrow} T \in \text{Arcs}(\Lambda)$

- **Weak Precondition:**

W1. A,B $\in$ Classes(Λ)

W2. C $\notin$ Classes(Λ)

W3. If A $\Longleftarrow$AA and B $\Longleftarrow$BB, then either AA $\overset{\star}{\Longleftarrow}$BB or BB $\overset{\star}{\Longleftarrow}$AA

- **Strong Precondition:**

  1. A $\overset{super}{\longrightarrow}$ D $\notin$ Arcs(Λ) for any D $\in$ Nodes(Λ)
  2. B $\overset{super}{\longrightarrow}$ D $\notin$ Arcs(Λ) for any D $\in$ Nodes(Λ)

  (Neither A nor B has a superclass)

- **Spawned Primitives:**

  S1. ADD CLASS C

  S2. ADD IS-A A C

  S3. ADD IS-A B C

  If either A or B has a superclass,

  S4. ADD IS-A C CC (where CC is either the superclass of A or of B. See weak precondition 3)

  BEGIN TRANSACTION

  S5. LIFT FIELD f SOURCE A TO C

  S6. LIFT FIELD f SOURCE B TO C†
  END TRANSACTION

  BEGIN TRANSACTION

  S7. LIFT OPERATION foo SOURCE A PARAMETERS (T1,T2) TO C

  S8. LIFT OPERATION foo SOURCE B PARAMETERS (T1,T2) TO C
  END TRANSACTION

- **Note:** If A and B extend classes AA and BB respectively, and AA is an ancestor of BB, then we can have C extend BB after factoring, and both A and B (and their descendents) will maintain access to all their former fields and operations. If there is no ancestor relationship between AA and BB this can't be done.

  The matching of operations is done by comparing signatures and return types.

- **Postcondition:**

  P1. C $\in$ Classes(Λ)

  P2. A $\overset{super}{\longrightarrow}$ C $\in$ Arcs(Λ), B $\overset{super}{\longrightarrow}$ C $\in$ Arcs(Λ)

  P3.

  a. C$\overset{f}{\longrightarrow}$ T $\in$ Arcs(Λ)

  b. A$\overset{f}{\longrightarrow}$ * $\notin$ Arcs(Λ)

c. B$\xrightarrow{f}$ * $\notin$ Arcs(Λ)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak preconditions W1, W2 and W3 are met.

  1. S1 is enabled by W2. Its postcondition (**C** $\in$ **Classes(**Λ**)**) meets P1.
  2. S2 and S3 are enabled by P1, W1 and W3. Their postconditions meet P2.
  3. S4 is enabled only in certain circumstances, as indicated in W3. S4 is never enabled under strong checking, since FACTOR's strong preconditions preclude A or B having a superclass. (See Figure 3.8)
  4. S5 is enabled even under strong checking, by P2, A1, and the fact that C doesn't yet have a field named f. Its postcondition meets P3a and P3b.
  5. S6 is executed under weak checking, even when strong checking is on for FACTOR. It is enabled by P2, A2, and P3a. Its postcondition meets P3c.
  6. S7 and S8 are never enabled under strong checking. Under weak checking, they are only enabled when the fields referenced and operations invoked by A::foo(T1,T2) and B::foo(T1,T2) are available at C, as indicated in Section 3.8.25.

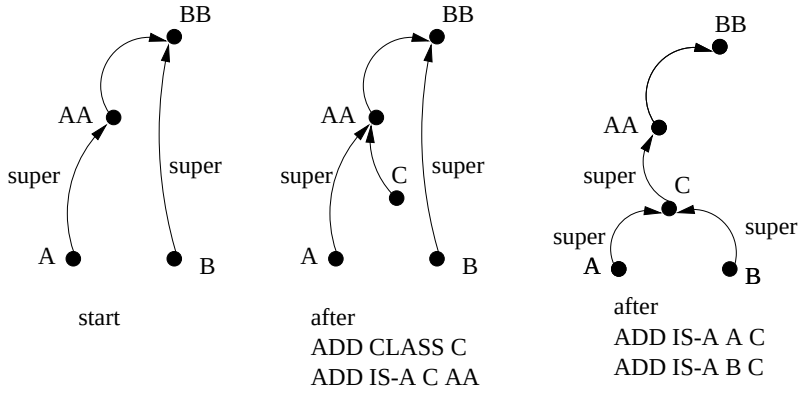

Figure 3.8: Factoring when input classes have superclasses

3.9.2 FLATTEN

- **Primitive:** FLATTEN *AbstractClassName*.
- **Example:** FLATTEN C, assuming:

  A1. A$\Longleftarrow$C

  A2. B$\Longleftarrow$C

  A3. C$\xrightarrow{f}$ T $\in$ Arcs(Λ)

  A4. C$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ)

- **Weak Precondition:** B $\in$ Classes(Λ).

- **Spawned Primitives:**

  BEGIN TRANSACTION

  S1. LOWER FIELD f SOURCE C TO A

  S2. LOWER FIELD f SOURCE C TO A
  END TRANSACTION
  BEGIN TRANSACTION

  S3. LOWER OPERATION foo SOURCE B PARAMETERS (T1,T2) TO A

  S4. LOWER OPERATION foo SOURCE B PARAMETERS (T1,T2) TO A
  END TRANSACTION

- **Postcondition:**

  P1. A$\xrightarrow{f}$ T $\in$ Arcs(Λ)

  P2. B$\xrightarrow{f}$ T $\in$ Arcs(Λ)

  Assuming only weak checking

  P3. A$^{foo(T1,T2)}\!\!\longrightarrow$ T $\in$ Arcs(Λ)

  P4. B$^{foo(T1,T2)}\!\!\longrightarrow$ T $\in$ Arcs(Λ)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak precondition is met.

  1. If A$\xrightarrow{f}$ U $\in$ Arcs(Λ) for some U $\in$ Nodes(Λ), P1 is met. Otherwise, A1 and A3 enable S1 using either weak or strong checking. S1's postcondition meets P1.
  2. If B$\xrightarrow{f}$ U $\in$ Arcs(Λ) for some U $\in$ Nodes(Λ), P2 is met. Otherwise, A2 and A3 enable S2 using either weak or strong checking. S2's postcondition meets P2.
  3. If A$^{foo(T1,T2)}\!\!\longrightarrow$ U $\in$ Arcs(Λ) for some U $\in$ Nodes(Λ), then according to the rules of Java, U = T, so P3 is met. Otherwise, assuming only weak checking is on, A1 and A4 enable S3. S3's postcondition meets P3.
  4. If B$^{foo(T1,T2)}\!\!\longrightarrow$ U $\in$ Arcs(Λ) for some U $\in$ Nodes(Λ), then according to the rules of Java, U = T, so P4 is met. Otherwise, assuming only weak checking is on, A2 and A4 enable S4. S4's postcondition meets P4.

- **Note:** If strong checking is on, S3 and S4 may or may not be enabled, according to whether any method called by foo(T1,T2) is overridden in the subclasses. See Section 3.8.27.

3.9.3 MERGE

- **Primitive:** MERGE $className_1$, $className_2$, ... INTO *NewClassName* USING *distinguishingAttribute*.

- **Example:** MERGE A, B INTO C USING status assuming:

A1. A$\xrightarrow{f}$ T $\in$ Arcs(Λ)

A2. B$\xrightarrow{f}$ T $\in$ Arcs(Λ)

A3. A$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ)

A4. B$\xrightarrow{foo(T1,T2)}$ T $\in$ Arcs(Λ)

- **Weak Precondition:**

  W1. A, B $\in$ Classes(Λ)

  W2. C $\notin$ Nodes(Λ)

  W3. Either A and B have no superclass, or it is the same superclass

  W4. Fields(A) = Fields(B); **Note:** matching on name and type

  W5. Fields(A) does not include a field named *status*

  W6. Operations(A) = Operations(B). **Note:** matching on signature and return type

- **Spawned Primitives:**

  S1. FACTOR A, B INTO C

  S2. ADD ATTRIBUTE status SOURCE C TARGET String

- **Postcondition:**

  P1. C $\in$ Classes(Λ)

  P2. A $\xrightarrow{super}$ C$\in$ Arcs(Λ), B $\xrightarrow{super}$ C $\in$ Arcs(Λ)

  P3.

  a. C$\xrightarrow{f}$ T $\in$ Arcs(Λ)

  b. A$\xrightarrow{f}$ * $\notin$ Arcs(Λ)

  c. B$\xrightarrow{f}$ * $\notin$ Arcs(Λ)

  P4. C$\xrightarrow{status}$ String $\in$ Arcs(Λ)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak precondition is met.

  1. W1, W2 and W3 meet the preconditions of S1. S1's postcondition satisfies P1 and P2.
  2. Since f is a matching field (A1, A2), it is lifted by FACTOR to C, entailing P3.
  3. If weak checking is on, the matching operation foo(T1,T2) (A3, A4) is lifted by FACTOR to C.
  4. W5 enables S2. S2's postcondition satisfies P4.

- **Note:** MERGE automates:

  1. The creation of a new merged class with a distinguishing attribute;
  2. The lifting of fields and operations to the merged class.

MERGE spawns LIFT OPERATION primitives for each matching operation, but under strong checking, these are never enabled. The designer who wishes to lift operations, will have to take the responsibility of manually checking that these lift operations are safe, and perform MERGE using only weak checking. To complete the merging of the input classes, the designer must also manually:

1. Replace invocations of input class constructors with invocations of the merged class constructor;
2. Migrate any existing instances of the input classes to the merged class after augmenting them with a *status* field;
3. Drop the input classes, by calling the DROP CLASS primitive.

3.9.4 SPLIT

- **Primitive:** SPLIT *singleClass* USING *distinguishingAttribute* VALUES "(" *Value1, Value2* ... ")". (Note: the *distinguishingAttribute* has type String, and Value1, Value2, etc. are strings composed only of letters, numerals, and underscores.)
- **Example:** SPLIT Patient USING status VALUES ("InPatient", "OutPatient").
- **Weak Precondition:**

  W1. Patient $\in$ Classes(Λ)

  W2. Patient $\xrightarrow{status}$ String $\in$ Arcs(Λ)

  W3. Patient_status_InPatient, Patient_status_OutPatient $\notin$ Nodes(Λ).

- **Spawned Primitives:**

  S1. ADD CLASS Patient_status_InPatient

  S2. ADD CLASS Patient_status_OutPatient

  S3. ADD IS-A Patient_status_InPatient Patient

  S4. ADD IS-A Patient_status_OutPatient Patient

  S5. FLATTEN Patient†

- **Postcondition:**

  P1. Patient_status_InPatient, Patient_status_OutPatient $\in$ Classes(Λ)

  P2. Patient_status_InPatient $\Longleftarrow$Patient,
  Patient_status_OutPatient $\Longleftarrow$Patient.

  P3. Fields(Patient_status_InPatient) =
  Fields(Patient_status_OutPatient) =
  Fields(Patient)

  P4. Operations(Patient_status_InPatient) =
  Operations(Patient_status_OutPatient) =
  Operations(Patient)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak precondition is met.

  1. W3 enables S1 and S2. Their postconditions satisfy P1.
  2. W1 and P1 combined with the fact that the newly added classes C_status_s1 and C_status_s2 do not extend other classes, enable S3 and S4. Their postconditions satisfy P2.
  3. W1 meets the precondition for S5, which may be safely executed under weak checking even when SPLIT is executed under strong checking. This is because the classes newly created by S1 and S2 cannot have any methods which override methods in Patient. The postcondition of S5, executed under weak checking, satisfies P3 and P4.

- **Note:** SPLIT automates:

  1. The naming and creation of new subclasses of the original class;
  2. The lowering of fields and operations to the subclasses.

  To complete the splitting of the original class, the designer must manually:

  1. Replace invocations of the input class constructor with invocations of the subclass constructors;
  2. Migrate any existing instances of the input class to the subclasses based on their values for the distinguishing attribute;
  3. Drop the input class, by calling the DROP CLASS primitive.

  SPLIT is useful when specialized fields and operations need to be added to the newly created subclasses, depending on the value of the distinguishing attribute.

3.9.5 AGGREGATE

- **Primitive:** AGGREGATE $className_1$, $className_2$... INTO *PartClassName* USING *hasaName*.

- **Example:** AGGREGATE Person, Company INTO Address USING address, assuming:

  A1. Person $\xrightarrow{city}$ String $\in$ Arcs(Λ)

  A2. Company $\xrightarrow{city}$ String $\in$ Arcs(Λ)

  A3. Person $\xrightarrow{getCity()}$ String $\in$ Arcs(Λ)

  A4. Company $\xrightarrow{getCity()}$ String $\in$ Arcs(Λ)

- **Weak Precondition:**

  W1. Person, Company $\in$ Classes(Λ)

W2. Address $\notin$ Nodes(Λ)

W3. W$\stackrel{address}{\longrightarrow} \notin$ Arcs(Λ) for any W $\in$ Refs(Λ) where Person$\stackrel{\star}{\Longleftarrow}$W or Company $\stackrel{\star}{\Longleftarrow}$W

- **Spawned Primitives:**

  S1. ADD CLASS Address

  S2. ADD HAS-A address SOURCE Person TARGET Address MULTIPLICITY 1

  S3. ADD HAS-A address SOURCE Company TARGET Address MULTIPLICITY 1
  BEGIN TRANSACTION

  S4. DELEGATE FIELD city SOURCE Person USING HAS-A address

  S5. DELEGATE FIELD city SOURCE Company USING HAS-A address
  END TRANSACTION
  BEGIN TRANSACTION

  S6. DELEGATE OPERATION getCity SOURCE Person PARAMETERS () USING HAS-A address

  S7. DELEGATE OPERATION getCity SOURCE Company PARAMETERS () USING HAS-A address
  END TRANSACTION

- **Postcondition:**

  P1. Address $\in$ Classes(Λ)

  P2. Person$\stackrel{address}{\longrightarrow}$ Address $\in$ Arcs(Λ)

  P3. Company$\stackrel{address}{\longrightarrow}$ Address $\in$ Arcs(Λ)

  P4. Address$\stackrel{city}{\longrightarrow}$ String $\in$ Arcs(Λ)

  P5. Person$\stackrel{city}{\longrightarrow}$ * $\notin$ Arcs(Λ)

  P6. Company$\stackrel{city}{\longrightarrow}$ * $\notin$ Arcs(Λ)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak precondition is met.

  1. W2 enables S1. S1's postcondtion satisfies P1.
  2. W3 enables S2 and S3. Their postconditions satisfy P2 and P3.
  3. A1, A2, P2, and P3, and the fact that the newly added class Address has no field named address, enable S4 and S5. Their postconditions satisfy P4, P5 and P6.
  4. S6 and S7 are spawned, but there is no guarantee that even their weak preconditions are met. However, in practice they might be met, say if the operation to be moved references only fields that were moved previously, and invoked no other operations.

3.9.6 DECOMPOSE

- **Primitive:** DECOMPOSE *partClassName*.
- **Example:** DECOMPOSE Address, assuming:

  A1. Address $\stackrel{city}{\longrightarrow}$ String $\in$ Arcs(Λ)

  A2. Address $\stackrel{getCity()}{\longrightarrow}$ String $\in$ Arcs(Λ)

  A3. Person $\stackrel{address}{\longrightarrow}$ Address $\in$ Arcs(Λ)

  A4. Company $\stackrel{getAddress()}{\longrightarrow}$ Address $\in$ Arcs(Λ)

- **Weak Precondition:** (Following the example above)

  W1. Address $\in$ Classes(Λ)

- **Spawned Primitives:**

  S1. RECLAIM FIELD city SOURCE Person USING HAS-A address

  S2. RECLAIM FIELD city SOURCE Company USING OPERATION getAddress

  S3. RECLAIM OPERATION getCity SOURCE Person PARAMETERS () USING HAS-A address

  S4. RECLAIM OPERATION getCity SOURCE Company PARAMETERS () USING OPERATION getAddress

- **Postcondition:** (Following the example above)

  P1. Person $\stackrel{city}{\longrightarrow}$ T $\in$ Arcs(Λ) for some T $\in$ Nodes(Λ)

  P2. Company $\stackrel{city}{\longrightarrow}$ U $\in$ Arcs(Λ) for some U $\in$ Nodes(Λ)

- **Reasoning About How the Postcondition is Attained:**

  Assume the weak precondition is met.

  1. If Person already has a field named city, P1 is met, otherwise A1 and A3 enable S1. S1's postcondition satisfies P1.
  2. If Company already has a field named city, P2 is met, otherwise A1 and A4 enable S2. S2's postcondition satisfies P1.
  3. S3 and S4 are spawned, but there is no guarantee that even their weak preconditions are met. However, in practice they might be met, say if the operation to be moved references only fields that were moved previously, and invoked no other operations.

3.10 Related Work

A framework for maintaining behavior and consistency of programs following schema evolution was presented by Hürsch and Seiter [55]. Consider an object-oriented system as a tuple <S,O,P> consisting of a schema S, an object store O, and a program P. A transformation τ consists of a tuple $< \sigma, \omega, \pi >$ where σ transforms the schema, ω transforms the object store, and π transforms the program. The intuitive goal is for τ to move the system from one correct state to another. That is, assuming that S and O conform to each other and that P produces a correct result R when executed on O, then $\sigma(S)$ and $\omega(O)$ should also conform to each other and $\pi(P)$ should also produce a correct result when executed on $\omega(O)$. If P operating on input I produces output R with the side effect that O is transformed to O', then $\pi(P)$ operating on $\omega(I)$ should produce output $\omega(R)$ and have the side effect that $\omega(O)$ is transformed to $\omega(O')$.

Data flow analysis is a standard compiler task, which enables determining the *liveness* of a variable at different points in the program. This requires relating uses of variables (on the right hand sides of expressions) to their definitions (assignments) using *def-use* chains [5]. Such chains are similar to the mappings between declarations and expressions developed here in Sections 3.4 and 3.5.

One elegant approach to def-use chains is known as *Static Single-Assigment* [36], or *SSA form*, which is an intermediate form in which each variable has only one definition in the program source. SSA form is described at length by Appel [5, 6].

IOM Paths and usage traces are related to path expressions which have received a formal treatment by Lausen and Vossen [66]. They deal with a simplified object-oriented language, for example, it allows a method name to be used only once in a class, no matter what the parameter list is.

Chapter 4

Transforming the IOM and Java Programs

4.1 Introduction

Chapter 2 described the Implementation Object Model (IOM), and the Change Specification Language (CSL). The current chapter is concerned with the mechanics of applying a list of change specifications to an existing set of Java source program files. A prototype program STP (*Schema Transformation Processor*) has been built to do just that. While the current prototype makes few claims as to robustness and efficiency, it has been tested with nearly all primitives included in the CSL, and clearly shows the feasibility of this approach.

One of the unique features of STP is that it builds, and extensively uses, a working version of the IOM. This *working IOM* abstracts the essential features of the declarations found in the Java source files. It is one of the two major data structures used, the other being the *abstract syntax tree* (AST) built by parsing the Java sources. Declarations in the working IOM have pointers to their usages in function bodies contained in the AST; these usages in turn have pointers to their declarations in the working IOM. Before applying CSL primitives, their preconditions are checked in the working IOM. Applying a primitive updates both the working IOM and the AST in synchrony, thus enabling a number of primitives to be applied, one after the other. Finally, the updated AST is printed out as Java source code.

The inputs to STP are a set of Java source files, and a list of CSL primitives to be applied. Option settings specify whether weak condition checking, strong condition checking, or no checking is desired. The first step is to parse the Java source code into an Abstract Syntax Tree (AST). This is described in Section 4.2.

Next there is a need to do some reverse engineering so as to build a working Implementation Object Model (IOM) from the AST. This is needed since the CSL is stated in terms of the IOM. The working IOM is a data structure containing lists of the ClassDef's, IS-A's and HAS-A's which are extracted by examining the AST. Each ClassDef further contains lists of its Attributes and Operations. This reverse engineering is described more fully in Section 4.3.

Next, each of the items in the working IOM is annotated with usage information. That is, a list is attached to each item, the list consisting of pointers into nodes of the AST where the item is used. This is done for two reasons: (1) Checking preconditions for transformations sometimes requires information as to whether certain items are actually used, (2) Implementing transformations such as renaming an IOM item requires that each usage of the item be renamed as well. The annotation is described in Section 4.4.

With the annotated working IOM in place, the system is ready to begin accepting change commands. A file of CSL commands is read and a CslCommand object is constructed from each command. The CslCommand objects are then processed by the CslCommand interpreter one-by-one. If type-soundness preservation is desired, the weak precondition is verified; if behavior preservation is desired, the strong precondition is verified as well. If the indicated preconditions are satisfied, the transformation is applied.

A Higher Level Primitive (HLP) is expanded into a sequence of Lower Level Primitives (LLP's) using the current IOM. These LLP's are then processed in order. For an LLP both the working IOM and the AST are updated to effect the transformation. The processing of LLP's is described in Section 4.5, and the expansion of HLP's in Section 4.6.

The whole procedure is illustrated in Figure 4.1.

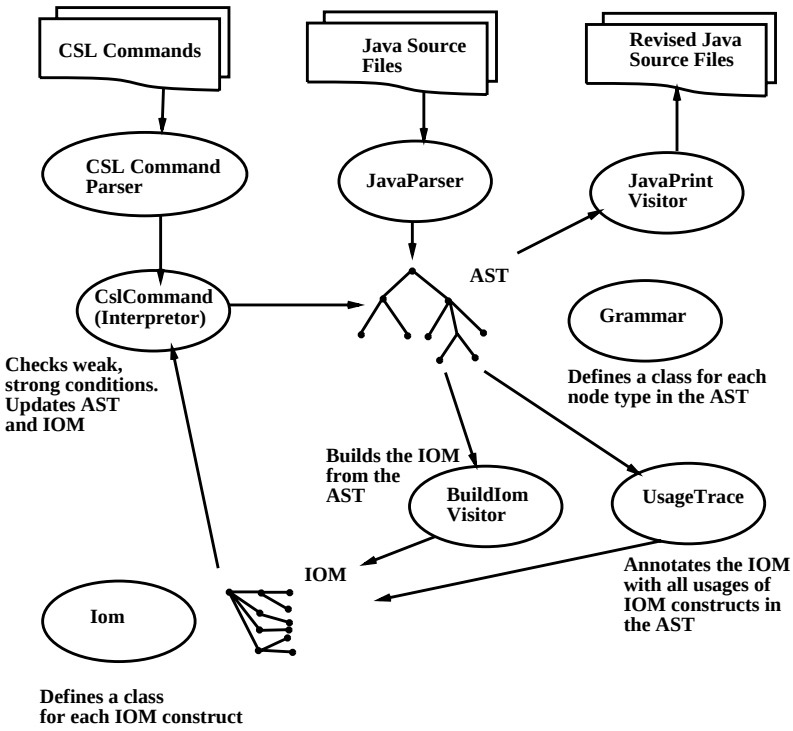

Figure 4.1: How STP works: Applying CSL commands to a set of Java source files

Section 4.7 details the individual tools that make up AST. Others have constructed similar prototypes, and some of these are discussed in Section 4.8.

4.2 Parsing Java Source Code

A recent tool, JavaCC, available from Sun Microsystems [102] greatly facilitates the task of parsing languages. An input file, usually with extension .jj (formerly .jack) is easily prepared from an Extended Backus Naur Form (EBNF) [91] grammar for the language. JavaCC then generates a recursive descent parser from this input. The parser can handle LL(k) grammars according to user-specified look-aheads in the input file.

To build the STP, a separate class was created for each non-terminal in the EBNF grammar for the Java language, Version 1.02. The class has fields for the items on the right hand side (RHS) of its defining production. When the RHS consists entirely of an alternation, an abstract class was used, with the classes representing the alternatives inheriting from it. Otherwise, if the RHS contains an alternation, as well as other items, fields for all the alternatives are provided. When a repetition appears on the RHS, a vector field is used.

Here are three examples of building classes from grammar productions:

1. ForInit := LocalVariableDeclaration | StatementExpressionList

```
class ForInit{
}
class LocalVariableDeclaration extends ForInit {
    ...
}
class StatementExpressionList extends ForInit {
    ...
}
```

2. PrimaryExpression := PrimaryPrefix (PrimarySuffix)*

```
class PrimaryExpression{
   private PrimaryPrefix primaryPrefix;
   private Vector primarySuffix;
}
```

3. BlockStatement := LocalVariableDeclaration ";" | Statement

```
class BlockStatement{
   private LocalVariableDeclaration localVariableDeclaration;
   private Statement statement;
}
```

The parser is provided with actions to construct an abstract syntax tree from the source file. The nodes are objects of the non-terminal classes as defined above.

4.3 Reverse Engineering to Build the IOM

Having captured the information from a set of Java source files into an abstract syntax tree, as above, the next step is to recover design information by reverse engineering the declarations found in the AST. This design information is represented in the working IOM.

4.3.1 Recovering Java Source Files From Class Files

It is possible to reconstruct Java source files from class files, using tools such as Mocha [39]. Java class files actually store the original names of classes, fields, operations and other named constructs as strings [73]. Providing the source files have not been deliberately obfuscated, sufficient design information can be recovered to construct the IOM. This is useful in cases where the original Java source files are not available. See Figure 4.2.

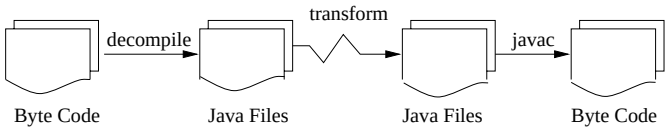

Figure 4.2: Transforming byte code in class files

4.3.2 Recovering Schema Design Information From Java Declarations

Recovering design information is essentially a process of locating declarations in the AST, and abstracting, that is removing details which have no counterpart in the IOM. Examples of such details are method bodies, constructors and destructors, and visibility modifiers.

For each class definition found in the AST, it is necessary to capture the name of the class, the class which it extends (if any), and information about its member fields and methods. For fields, the name, type and multiplicity are used. For methods, the signature and return type are used. Interfaces are not considered.

If a field is an array, or has a type such as Vector, or another collection type, it is considered to be multiple-valued, otherwise single-valued. However, there is a problem in recovering underlying type information when a field's type is a collection type. In Java, all classes extend a built-in class named Object [7]. This is used in utility collection classes such as Vector and Hashtable. For example, suppose a programmer needs to maintain a collection of Book objects for a lending library. The Books could be kept in a Hashtable, hashed by ISBN number. When an object is returned from the Hashtable it needs to be cast to a Book object before Book operations can be used. The reverse engineering problem is that the Java class definition for say, Library, will show a field, say books, of type Hashtable. It is not clear what the type is for the objects stored in the Hashtable; they may even be different types. In the IOM, the field Library.books will become a HAS-A link since a Hashtable is a collection, the link will have multiplicity n; the source of the link is Library, what is missing is the target of the link.

One solution is to attempt to infer a least common supertype for the objects stored in a collection. Following the type inference methods used by Palsberg and Schwartzbach [85], type variables are associated with these objects. Constraints on the variables are derived from their usages in constructor calls, casts, etc. Solving these constraints may allow determining a common supertype for the objects which is more specific than Object. It may even be possible to determine a least common supertype, or the actual type (when the objects in the collection have the same type).

The second solution is to discourage the use of generic collections altogether, replacing them by parameterized types or templates as used in C++. Although parameterized types do not currently exist in Java, there are several proposals for introducing them [77, 80, 22]. The present version of STP solves the problem simply, by limiting multiple-valued fields to be arrays. When, and if, parameterized types become standard in Java, it will be easy to include them as well, and presumably there will be less use made of generic collection types.

4.4 Annotating the Working IOM with Usage Information

The working IOM is a data structure containing lists of the ClassDef's, IS-A's and HAS-A's. Each ClassDef further contains lists of its Attributes and Operations. A program, *usageTrace* has the job of annotating each item in an IOM list with usage information. That is, a list is attached to each IOM item, consisting of pointers into nodes of the AST where the item is used.

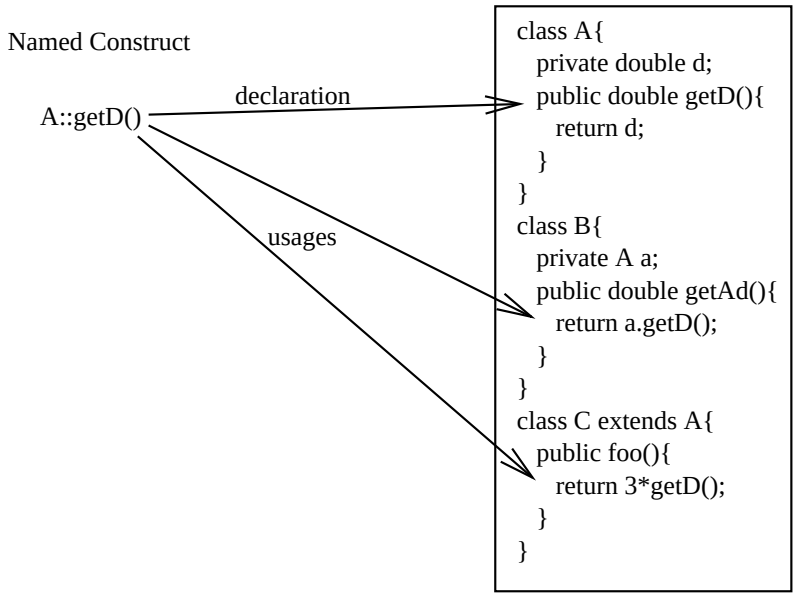

Figure 4.3: A Named Construct with pointers to its declaration and usages in the program. (Actually the pointers are into the abstract syntax tree built by parsing the program.)

Figure 4.3 shows an IOM operation object (a kind of named construct), with a pointer to its declaration, and a list of pointers to its usages. The working IOM can be described succinctly using a notation that will be called Cb (for Class Builder), patterned after the cd notation used in Demeter. To understand Cb notation, consider the sentence:

```
Operation : NamedConstruct =
    <returnType>         IomType
    <parameterTypes>*    IomType
    <methodDeclaration>  MethodDeclaration  //ptr into AST
.
```

This sentence defines class: Operation. The ":" in "Operation : NamedConstruct" indicates that Operation inherits from NamedConstruct. The fields of the class are enclosed in angle brackets with their type to the right. A "*" indicates a multiple-valued field. Hence, Operation defines three fields: (1) A single-valued field of type IomType named returnType; (2) a multiple-valued field of type IomType named parameterTypes; and (3) a single-valued field of type MethodDeclaration named, appropriately, methodDeclaration.

The Working IOM in Cb Notation

```
Iom =
    <classDefs>*            ClassDef
    <isas>*                 Isa
    <hasas>*                Hasa
    <restrictions>*         Restriction
    <compilationUnit>       CompilationUnit  //ptr into AST
.
NamedConstruct =
    <name>              Identifier
    <usages>*           Usage   //pointers to nodes in AST where name is used
.
IomType : NamedConstruct =
.
ClassDef : IomType =
    <attributes>*       Attribute
    <operations>*       Operation
    <classDeclaration>  ClassDeclaration  //ptr into AST
.
Attribute : NamedConstruct =
    <type>              PrimType
    <fieldDeclaration>  FieldDeclaration  //ptr into AST
.
//  A PrimitiveType is a primitive such as int, long, float, char, String, etc.
PrimType : IomType =
.
Operation : NamedConstruct =
    <returnType>         IomType
    <parameterTypes>*    IomType
    <methodDeclaration>  MethodDeclaration  //ptr into AST
.
Isa =
    <subClass>          Identifier
    <superClass>        Identifier
.
Hasa : NamedConstruct =
    <sourceClass>       Identifier
    <targetClass>       Identifier
    <multiplicity>      int
```

```
    <inverseLinkName>   Identifier
    <fieldDeclaration>  FieldDeclaration  //ptr into AST
.
Restriction =
    <predicate>        String
.
Usage =
    <identifier> Identifier
    <primaryExpression> PrimaryExpression
.
```

UsageTrace acts as a visitor doing a depth-first traversal of the AST. Its goal is to identify names found in expressions with named items in the working IOM, then to set pointers from the IOM items to the AST nodes where the items are used. Back pointers are also set from the AST nodes to the IOM items. This task is complicated by the scoping and inheritance rules of the language used, and by the difficulty of identifying lexical elements in the language with what are essentially design elements in the IOM.

The usages of named items such as attributes, HAS-A's and operations are found by examining the expression syntax of the language. In particular, so-called *path expressions* such as s.getAdvisor().getCourses().names may contain multiple usages. However, identifying the usages requires care. The key in this example is to identify the start of the path s using scope information. It may be a variable or a HAS-A. In either case, once s is identified its type can be determined, which allows the identification of getAdvisor(). Next, getCourses() and then names can be identified as well. This was discussed at greater length in Section 3.5.

4.5 Applying LLP Transformations

In the current version of STP, it is assumed that the user will want at least the weak condition checked before applying the LLP. This may be relaxed in future to accommodate users who wish to to force a transformation, disregarding contra-indications.

LLP's are applied as follows:

1. The weak condition for the LLP is checked. Each CSL command object has an operation: meetsWeakConditions(). This operation checks in the IOM to make sure the various classes, fields, operations, inheritance relationships, etc. specified in the LLP actually exist. When it finds these items it caches them in the CSL object itself for use by the follow-up operations. If any required item is not found, or if any other weak condition associated with the LLP is not satisfied (see Section 3.7), this function returns immediately with the value false, stopping further processing of the LLP.

2. If strong checking is indicated, the meetsStrongConditions() operation is invoked. If a strong condition is not satisfied it returns false, stopping further processing of the LLP.

3. The IOM is updated by the function updateIom(), to reflect the transformation.
4. The abstract syntax tree is updated by the function updateAst(), to reflect the transformation.

The functions updateIom(), and updateAst() must work in synchrony, so that following the update, the IOM contains an accurate description of the schema implied by the AST. Care must be taken so that usages of named constructs are correctly reassigned following the update. In this way subsequent transformations can be applied, without having to rebuild the IOM from scratch.

4.6 Expansion of HLP's

The CSL command interpreter processes an HLP as follows:

1. The weak condition of the HLP is checked.
2. If strong checking is required, the strong precondition is checked.
3. The HLP is expanded into a list of LLP's, as specified in Section 3.9.
4. The HLP is removed from the pending CSL operations, and replaced by the list of its spawned LLP's.

The file of original CSL's and the file of expanded CSL's created during update can be kept providing a historical record of changes to the schema. This history can provide some valuable semantic information. For example, schema integration requires detection of synonyms and homonyms [65]. Knowing schema version histories can help decide these questions. See [67].

4.7 Tool Summary

Here is a summary of the tools used by STP, as pictured in Figure 4.1.

1. Parsers
   (a) CslCommandParser - Parses a command written as a CSL sentence into a CslCommand object.
   (b) JavaParser - Builds an abstract syntax tree from a list of Java source files, in accordance with the Java grammar used.
2. Generators
   (a) BuildIomVisitor - A visitor object which traverses an abstract syntax tree, extracting declarations, and building a working IOM model from it.

(b) UsageTrace - A visitor object which traverses an abstract syntax tree, identifying primary expressions with paths in the IOM graph view of the working IOM. It adds pointers to expression components as usages of IOM constructs, and back pointers from expression components to their declarations in the IOM.

(c) JavaPrintVisitor - A visitor object which traverses an abstract syntax tree representing a Java program, and generates Java source code from it.

3. Transformers

(a) CslCommandinterpreter - Given an AST, and an IOM generated from it, and a list of CSL transformation objects, apply the transformations to both expanding HLP's as needed. Weak preconditions and possibly also strong ones are verified, as specified in options.

4.8 Related Prototype Development

There has been some work on developing prototypes in support of schema transformations. Opdyke [81], as part of his thesis, prototyped some of his refactorings in C++, however, apparently this was limited owing largely to the complexity of the C++ grammar. Brant [95] et. al carried over many of these refactorings into a browser tool for Smalltalk.

Womble [56] is a tool recently developed at MIT, that extracts object models from Java bytecode. Womble's model is similar to IOM Graphs, in that its nodes represent Java classes, and its edges represent either subclassing or associations. However the model is different in appearance, following the OMT style.

The chief innovation of Womble is its treatment of container classes such as hash tables or vectors. Womble attempts to infer the underlying types of the objects stored in such containers, and is reportedly successful most of the time. This feature should prove to be extremely useful to the STP tool. One of the stumbling blocks at the present time, is that after importing a Java file into STP, it is necessary to manually replace references to container objects by multi-valued references to their underlying classes. Womble technology may well complement and enhance existing STP methods for extracting object models.

Chapter 5

The Itinerary Language

5.1 Introduction

This section introduces the Itinerary language and some of its transformations.

In a shared, persistent object system, the classes often represent repositories of data and methods which are made use of in a variety of different applications. Each individual application requires the collaboration of several classes. Holland [51, 50] borrowed the language of *contracts* to capture the roles which participant classes played in the application. Riehle and Gross [94] extend these ideas into *role modeling* for the purpose of framework design, as opposed to class-based modeling. Mezini and Lieberherr [75] also describe collaborations which cut across classes, in terms of *adaptive plug-and-play components.*

It is the point of view here that application programs are built to serve the needs of end-users, and that these end-users can be represented within the system as objects. These objects match up with the so-called *external entities* of traditional data flow diagrams [38], in that they are the sinks and sources of data flows. For example, supposing in a University environment, an application is built to provide students with a copy of their own transcripts. *Student* would be a class in the system, and each individual student an object in that class. The flow of the application can be visualized as originating with a specific student object, and traversing through the run-time object system, visiting classes such as *Course*, *Program*, *Section*, etc. to gather the information needed, finally returning a report to the Student object. This brings to mind the travel metaphor of an itinerary, or sequence of stops with connecting legs between them.

Programming such an application can be broken down into two tasks, namely (1) fixing on the itinerary, and (2) deciding on the work that needs to be done at each of the stops. Separating these tasks allows for the possibility that the itinerary can be re-used; in the University example the same itinerary needed to create a student transcript could be re-used to create a current class schedule for that student. This *separation of concerns* is one of the features of adaptive programming [69, 71, 87]. Itineraries can also be seen as an application of the *Visitor Design Pattern* [44].

The possibility of programming with itineraries raises a number of questions:

1. What support can be offered in specifying itineraries? This is addressed in Section 5.2.
2. How can itinerary specifications be combined with tasks to be done at stops so as to build an application? Addressed in Section 5.3.
3. How can new itineraries be attached to an existing object system? Addressed in Section 5.4.
4. Since a system can be viewed as the *sum of its applications*, can a complete system be generated by combining the itineraries it needs to support? See Section 5.5.
5. How can itineraries be used to limit access for security purposes at the objects visited? Addressed in Section 5.6.
6. Can itineraries be specified in a general way, and then mapped to specific classes and links? Addressed in Section 5.7
7. When schema transformations are applied, as in Chapters 2, 3 and 4, can the itineraries themselves be automatically updated to conform to the new schema? Addressed in Section 5.8.

A visual representation of program declarations in the form of Extended IOM Graphs was introduced in Chapter 3. Some of the class nodes act as surrogates for external entities (*end-users*). This can be seen in Figure 5.1, where the nodes labelled Professor, Student, and Health Officer represent end-users who can expect to get reports such as class lists, personal schedules, and health records, respectively, from the system.

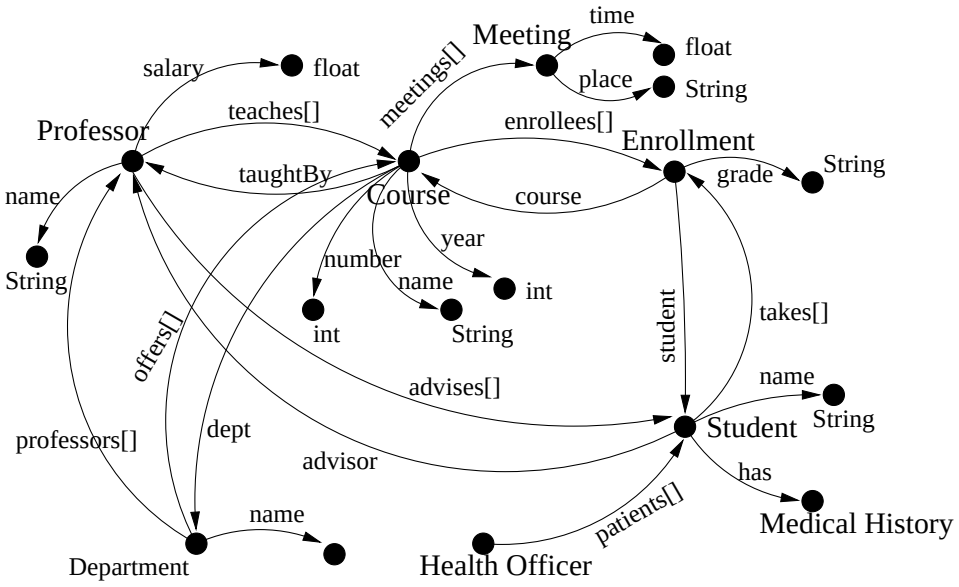

Figure 5.1: An extended IOM Graph for a university

The applications which produce these reports can be envisioned as traversals through the graph. For example, the class list report follows the itinerary:

$$Professor \overset{teaches[]}{\longrightarrow} Course \overset{enrollees[]}{\longrightarrow} Enrollment \overset{student}{\longrightarrow} Student$$

However, much of the information stored in the system is private, meaning that not all paths shown are permitted to be traversed. For example, Professors ought not have access to a student's medical history, nor students to other students grades.

Seen in terms of an Extended IOM Graph, an itinerary is composed of *legs* which correspond to HAS-A arcs. In addition, source and destination preconditions can be specified to be checked before and after traversing a leg, respectively. These preconditions can be used to limit access.

As in the visitor design pattern [44], work is done in the course of an itinerary by visitor objects carried by the traversals. The visitors have *before* and *after* methods for various nodes. Each leg traversal is implemented by a function call such as Student::getAdvisor(). In the case of multi-valued legs, an iterator is invoked. The visitor's before method is called on first arriving at a node, with the node itself passed to the visitor as a parameter. The after method is called on returning from the node. Itineraries can be built up from legs by joining leg traversals end-to-end, or by unioning them together at nodes to form branches.

The itineraries available to an end-user form a kind of database view, indicating paths and access privileges which are available to her applications. Such object-oriented views are analogous to views in the three-level ANSI/SPARC architecture [108, 40].

The shared system itself can be designed in bottom-up fashion by first identifying end-users and their use cases [58], creating itineraries for each, and then combining the itineraries. See Section 5.5.3. Finally, the itineraries can themselves be transformed, using the same change specification language as used for Java source files.

5.1.1 Itineraries and Adaptive Programming

The idea for Itineraries originated when trying to build database programs using the methods of Adaptive Programming referred to in Section 1.4.2. Some academic work has been done in this area [49], however, a full-fledged business application requires a schema which is richly connected, typically with multiple paths between any two objects. For example, ODMG [29] calls for inverses to each association. Following this standard implies that even adjacent objects would have infinitely many paths between them. The path completion algorithms used by adaptive programs must be carefully constrained, when operating in such schemas. This negates much of the benefit of having adaptive programming in the first place; the constraints contain extensive schema knowledge and must themselves be updated and/or added to after many kinds of schema change. Applications such as compilers, where the most prevalent data structure is a tree, are good fits for adaptive programming techniques; other applications such as business systems involving large shared databases are seen to be less suitable when the costs of constraining the automatic path completions are balanced against the benefits of adaptiveness.

Like adaptive programming, itineraries allow for a concise description of traversals through a class structure. A tool attaches itineraries to existing classes, by adding operations needed to support the itinerary traversals. Itineraries also allow for the specification of preconditions to be evaluated before following each edge and after arriving at each node. Itinerary descriptions can be *concrete*, meaning there is a one-one mapping from itinerary legs into

HAS-A links, or *abstract*, meaning that a single itinerary leg can map into a set of paths of HAS-A links. It is also possible to create a system of class definitions in the first place, starting from a list of itineraries annotated with type and multiplicity information.

5.2 Specifying Itineraries

The fundamental unit used in specifying an itinerary is the Leg, which attaches to a HAS-A link in the underlying class graph. A leg has 4 fields separated by commas:

[*Source Class Name*, *HAS-A Name*, *Source Condition*, *Destination Condition*]

The first two fields identify the HAS-A to attach to, the third field a condition to be evaluated at the source class of the HAS-A, the fourth field a condition to be evaluated at its destination.

An example of a leg, which is a part of an itinerary named *toGradStudents*, built over the Iom Graph in Figure 5.1 is:

$$[Professor, teaches, , host.number > 1000]$$

This leg is along the multi-valued HAS-A link: Professor $\stackrel{teaches[]}{\longrightarrow}$ Course. There is no source condition. On reaching each of the Course objects, the destination condition, *host.number > 1000*, is evaluated. If true, processing of the itinerary proceeds, otherwise it aborts, with no further work being done at the destination object, or in legs which follow from it.

Itinerary legs can be combined using *joining* and *unioning*.

- *Joining* is indicated by concatenating legs as in [A,h, .., ..][B,g, .., ..], where the target of leg [A,h, .., ..] is either B, or a subclass of B.
- *Unioning* is indicated by using the + operator and enclosing the unioned legs in parentheses as in ([B,i, .., ..]+[B,j, .., ..]+ ..). The source class of the unioned legs must be the same.

A leg can also be joined to a union using concatenation as in [A,h, .., ..]([B,i, .., ..]+[B,j, .., ..]), where the target of [A,h, .., ..] is either B, or a subclass of B.

An *itinerary join* consists of zero or more joined legs followed by an optional join to an itinerary union. Unioning is extended to allow itinerary joins to be unioned.

More formally:

- Itinerary ::= “ITINERARY” Name ItineraryJoin
- ItineraryJoin ::= {Leg}* [ItineraryUnion]
- ItineraryUnion ::= “(” ItineraryJoin {“+” ItineraryJoin}* “)”

- Leg ::= "[" ClassName "," HasaName ["[]"] [":" ClassName] "," [SourceCondition] "," [DestinationCondition] "]"

Here Name, ClassName and HasaName are simply identifiers, and the optional SourceCondition and DestinationCondition are Java expressions which may contain references to members of the host (source or destination classes respectively), as well as members of a visitor object, and evaluate to true or false. Following the HasaName are optional annotations "[]" and ":"ClassName, which can be used to indicate a multiple-valued HAS-A, and the target type of the HAS-A, respectively. These annotations are used when creating a new set of class definitions from the itinerary.

When an itinerary is implemented, each itinerary join is traversed left to right by making function calls. When the calls return, the traversal is retraced right to left. An itinerary union is traversed in the order of the unioned itinerary joins.

Figure 5.2 shows the itinerary *toGradStudents*. It is used for traversing to all students currently enrolled in graduate classes taught by a professor. It describes a traversal along the path:

$$Professor \overset{teaches[]}{\longrightarrow} Course \overset{enrollees[]}{\longrightarrow} Enrollment \overset{student}{\longrightarrow} Student$$

in the extended IOM graph in Figure 5.1.

```
ITINERARY toGradStudents
//used by a professor to access her current students in graduate level
classes
[Professor, teaches, ,host.number > 1000]
[Course, enrolls, host.year == 1999, ]
[Enrollment, student, , ]
```

Figure 5.2: Traversal from a Professor object to her current students in graduate classes

Since Professor::teaches and Course::enrolls are multi-valued HAS-A's, the code generated to traverse them must contain iterators. To carry out a task such as building class lists for graduate students, a visitor object is carried along by the traversal. Each itinerary has associated with it an abstract visitor class. Its name is constructed from the itinerary name by appending "_Visitor".

The Source and destination conditions are evaluated before or after traversing a leg, respectively. These predicates can refer to members of the associated visitor class, as well as the host (the class being visited). Source conditions are particularly useful in the case of itinerary unions, allowing a different precondition to be attached to each join in the union.

The abstract itinerary visitors must contain all visitor members used in these conditions. In addition, they contain before and after methods for each host being visited. The before method is called upon first arriving at an object by traversing a leg, the after method just prior to returning on that leg. Additional before and after methods are specified for the itinerary as a whole; the host for these is the *root class* of the itinerary, which is defined as the

source of the initial leg in the itinerary, where the itinerary consists of an ItineraryJoin with at least one leg, or the common source of the joins in the initial ItineraryUnion otherwise.

Thus the visitor toGradStudents_Visitor has at least the following interface:

```
class toGradStudents_Visitor{
  public void before(Professor host);
  public void before(Course host);
  public void before(Enrollment host);
  public void before(Student host);
  public void after(Professor host);
  public void after(Course host);
  public void after(Enrollment host);
  public void after(Student host);
}
```

A concrete visitor class may add additional members to those required by the conditions, and may override some of the before and after methods, so as to accomplish its task.

The order of processing of this itinerary is as follows:

1. The visitor's before method for the Professor class is executed, perhaps doing some initialization.
2. The source condition (absent in this example) is evaluated at the Professor object. If false no further traversal is made,
3. The reference edge, teaches, is followed to the target object(s) by making a function call; since this edge is multi-valued an iterator is invoked.
4. The destination condition, in this case "host.number>1000" is evaluated at each Course object reached. When false the object is skipped.
5. The visitor's before method for the Course class is executed.
6. The source condition: "host.year == 1999" for the second leg, is evaluated at the Course object. If false, control reverts to step 12. (This condition could have been combined with the condition in step 4.)
7. An iterator is invoked to traverse the HAS-A Course::enrolls to Enrollment objects.
8. The visitor's before method for Enrollment is invoked.
9. The Enrollment::student HAS-A is traversed.
10. The visitor's before and after methods for Student are executed.
11. The visitor's after method for Enrollment is executed.
12. The visitor's after method for Course is executed.

13. The visitor's after method for Professor is executed.

The next example, Figure 5.3 lists a person's in-laws. An in-law is narrowly defined here to mean either the sibling of a spouse or the spouse of a sibling. This example shows an itinerary with cycles in the class graph.

```
ITINERARY inlaws
( [Person,spouse, , ][Person,siblings, , ]
+ [Person,siblings, , ][Person,spouse, , ]
)
```

Figure 5.3: An itinerary to list a person's in-laws

A class graph supporting this itinerary is shown in Figure 5.4.

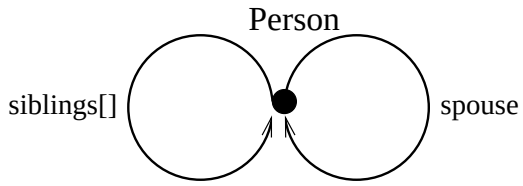

Figure 5.4: View projected by the in-laws itinerary

5.3 Programming With Itineraries

The execution of an object-oriented program can be visualized as consisting of a sequence of actions, including constructing and destroying objects, setting and obtaining field values including HAS-A fields that connect objects together, exercising methods, etc. Generally, each action targets a particular object in the system. For persistent systems, such as object databases, there is already a pool of available objects at the time the program starts up; for short-lived systems, all objects are created and destroyed within a program's lifetime.

Bachman, in his 1973 ACM Turing Award Lecture [8], makes the salient point that the availability of direct access storage devices *is changing the programmer from a stationary viewer of objects passing before him in core into a mobile navigator who is able to probe and traverse a database at will.* In an object system, it is the HAS-A links which provide the paths for navigating from one object to another. Itineraries, by describing paths of HAS-A links, are a good vehicle for encapsulating the navigational behavior of a program.

The *Law of Demeter* [70], a style guideline introduced by Lieberherr, requires that a method which is a member of class C should invoke only the methods of the following kinds of objects:

1. C
2. the parameters to the method

3. any objects it creates/instantiates

4. its direct component objects

A method may not directly invoke methods reached only by dereferencing a path of components. This would be too brittle, breaking if the component relationships were to undergo change. Instead, a direct component should be asked to invoke the method, which would then ask one of *its* direct components, and so on, until the object implementing the method is reached. Again, the metaphor is a navigational one.

The *Visitor Design Pattern* [44], is a good example of a programming method that separates navigational from behavioral concerns. When programming with this pattern, domain classes (which represent entities in the domain of discourse), are *light-weight* in the sense that they provide a repository for local state information and support for navigation by a visitor. Complex behavior that requires the cooperation of a number of different classes is encapsulated in a single visitor class, an instance of which is passed around to the various domain objects concerned. This approach reduces the clutter in domain classes, and allows for easy addition of new behaviors by subclassing the Visitor class. There are two class hierarchies, one for domain classes, and the other for visitors.

Programming with itineraries utilizes the visitor pattern. When an itinerary is attached to an existing Java system, it automatically generates support for carrying a visitor along the itinerary, and also generates an abstract visitor class, associated with the itinerary, which has before and after methods for each class visited.

5.4 Attaching Itineraries

An itinerary can only be attached to an existing system of Java class definitions if it is *compatible* with it. Let S be a set of Java class definitions, let Classes(S) be the set of names of classes in S, and denote members of classes, in C++ style, using the scope resolution operator "::". For example: C::x for a field, and C::foo(int, Y) for an operation, where $C, Y \in Classes(S)$.

An itinerary I is *compatible* with a system S of Java class definitions if:

1. For each class name C that appears in any leg in I, either as a source class or an annotation:
$$C \in Classes(S)$$

2. For each leg [A, h, .., ..] in I, A::h is a member field (possibly inherited) of A with type T, where $T \in Classes(S)$.

3. For each leg [A, h, .., ..] in I, if h is annotated with "[]", then A::h is multi-valued; if h is annotated with ":" B, then A::h has type B.

4. For each pair of joined legs [A, h, .., ..][B, i, .., ..] in I, the type of A::h is B or a descendent class of B.

5. For each instance of a leg followed by a union [A, h, .., ..] ([B, i, .., ..] + [B, j, .., ..] + ...), the type of A::h is B or a descendent class of B.

6. For each leg [A, h, E, ..] in I, with a source condition E, if E contains any usages of member fields or operations, other than members of visitors, these members must be defined in A or inherited into A.

7. For each leg [A, h, .., F] in I, with a destination condition F, if F contains any usages of member fields or operations, other than members of visitors, these members must be defined in or inherited into class T, where T is the type of h.

Attaching an itinerary to an existing Java system involves:

1. Checking that the itinerary is compatible with the Java system.

2. Creating support for traversal by an itinerary visitor, by attaching methods at each class involved.

3. Creating an abstract visitor class, specific to the itinerary, with before and after methods for each class visited.

The following kinds of methods are attached to support an itinerary named "Yy":

1. At the root class of Yy, a method:

   boolean Yy(Yy_Visitor v);

   This method is used to launch the itinerary. The root class is typically a surrogate for an actor, on whose behalf the itinerary is being built. An actor object can create a Yy_Visitor, and pass it to Yy. Yy(Yy_Visitor) calls the visitor's before method, passing it the root object, then calls the TakeOff method for each leg for which the root is a source, and finally the visitor's after method.

2. At each class X which is the source of a leg along HAS-A h, a method:

   boolean Yy_via_h_Takeoff(Yy_Visitor v);

   The Takeoff method checks the source condition for the leg; if it is true the method navigates along the HAS-A, invoking an iterator if multi-valued, and calls the Land methods for the leg at each destination.

3. At each class Y which is the destination of a leg along HAS-A X::h, a method:

   boolean Yy_from_X_via_h_Land(Yy_Visitor v);

   The Land method checks the destination condition, and if true, calls the visitor's before method, invokes the Takeoff methods of any following legs, and calls the visitor's after method.

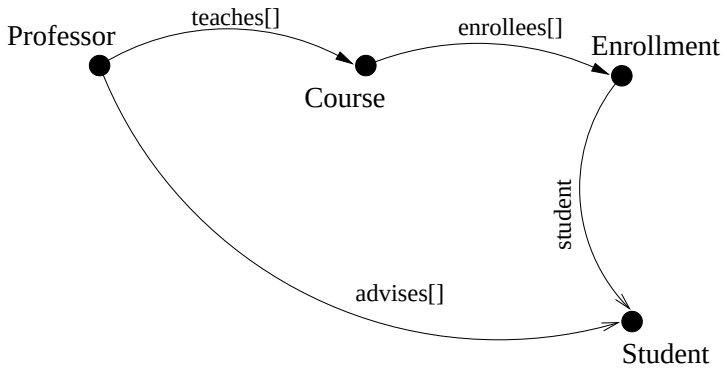

Figure 5.5: An existing Iom graph, compatible with the toStudents itinerary

The following example illustrates the methods added by attach. Consider the class graph in Figure 5.5, and the itinerary:

```
ITINERARY  toStudents
(
[ Professor,  teaches, , ]
[ Course,  enrollees, , ]
[ Enrollment,  student, , ]
 +
[ Professor,  advises, , ]
)
```

The code shown below is attached.

```
class Professor{
    public boolean toStudents(toStudents_Visitor v){
       boolean ret =  true;
       v. before (this );
       ret = ret &&  toStudents_via_teaches_TakeOff ( v );
       ret = ret &&  toStudents_via_advises_TakeOff ( v );
       v. after (this );
       return  ret;
   }

    public boolean toStudents_via_teaches_TakeOff(toStudents_Visitor v){
       boolean ret =  true;
       if ( teaches == null)
         {
             System. out. println ("Hasa teaches doesn't exist" );
             return  false;
         }
      for (int i =  0; i < teaches.length;  i++)
         {
             if (teaches[i] != null)
               {
```

```
                teaches[i].toStudents_via_teaches_Land ( v );
              }
         }
      return  ret;
   }
    public boolean toStudents_via_advises_TakeOff(toStudents_Visitor v){
       boolean ret =  true;
       if (advises == null)
         {
            System. out. println ("Hasa advises doesn't exist" );
            return  false;
         }
      for (int i =  0; i < advises.length; i ++)
         {
             if (advises[i] != null)
               {
                   advises[i].toStudents_via_advises_Land ( v );
               }
         }
      return  ret;
   }
}

class Course{
    public boolean toStudents_via_teaches_Land(toStudents_Visitor v){
       boolean ret =  true;
       v. before (this );
       ret = ret &&  toStudents_via_enrollees_TakeOff ( v );
       v. after (this );
      return  ret;
   }
    public boolean toStudents_via_enrollees_TakeOff(toStudents_Visitor v){
       boolean ret =  true;
       if ( enrollees == null)
         {
            System. out. println ("Hasa enrollees doesn't exist" );
            return  false;
         }
      for (int i =  0; i < enrollees.length; i ++)
         {
             if (enrollees[i] != null)
               {
                   enrollees[i].toStudents_via_enrollees_Land ( v );
               }
         }
      return  ret;
   }
}

 class  Enrollment{
    public boolean toStudents_via_enrollees_Land(toStudents_Visitor v){
       boolean ret =  true;
```

```
        v. before (this );
        ret = ret &&  toStudents_via_student_TakeOff ( v );
        v. after (this );
        return  ret;
    }
     public boolean toStudents_via_student_TakeOff(toStudents_Visitor v){
        boolean ret =  true;
        if (student == null)
          {
              System. out.println ("Hasa student doesn't exist" );
              return  false;
          }
        ret = ret &&  student.toStudents_via_student_Land ( v );
        return  ret;
    }
}

class Student{
     public boolean toStudents_via_student_Land(toStudents_Visitor v){
        boolean ret =  true;
        v. before (this );
        v. after (this );
        return  ret;
    }
     public boolean toStudents_via_advises_Land(toStudents_Visitor v){
        boolean ret =  true;
        v. before (this );
        v. after (this );
        return  ret;
    }
}
```

5.5 Itineraries to Java Code

In addition to being able to attach an itinerary to an existing class system, it is possible to create a set of class definitions in the first place, by automatically generating them from a list of itineraries. The process of combining itineraries and producing a set of Java source files is illustrated in Figure 5.6. The starting point is a list of itineraries, annotated with type and multiplicity information. An example is:

```
ITINERARY  toStudents
(
[ Professor, teaches[]:Course, , ]
[ Course,  enrollees[]:Enrollment, , ]
[ Enrollment,  student:Student, , ]
 +  [ Professor,  advises[]:Student, , ]
)
```

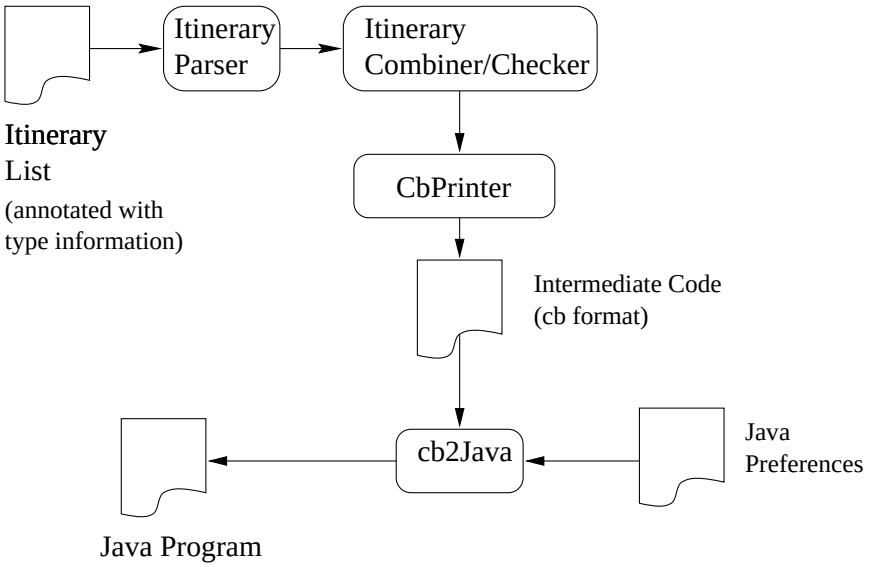

Figure 5.6: Tools for Combining Itineraries into a Java Program

```
ITINERARY  mySchedule
[ Student,  takes[]:Enrollment, , ]
[ Enrollment,  course:Course, , ]
[ Course,  meetings[]:Meeting, , ]
```

The list of itineraries is fed into an itinerary parser, and then into a combiner/checker, The combiner/checker adds new classes and members as they are discovered in itineraries, and looks for conflicts, such as member fields of a class with the same name but different types or multiplicities.

The next step, done by CbPrinter, is to print out the combined classes in an intermediate language called Cb (for Class builder). Cb was introduced in Section 4.4. Here is the output from the above itinerary list, in Cb form.

```
Professor =
   <teaches>* Course
   <advises>* Student
.
Course =
   <enrollees>* Enrollment
   <meetings>* Meeting
.
Enrollment =
   <student> Student
   <course> Course
.
Student =
```

```
   <takes>* Enrollment
.
Meeting =
.
```

In Cb notation, each sentence represents a class definition, names of member fields are shown in angle brackets, and their types are shown to their right. An asterisk indicates a multi-valued field. It is also possible to indicate member operations, as in:

<foo (int, X)> int

which is equivalent to the C prototype:

int foo(int, X);

The Cb file above is equivalent to the Iom graph in Figure 5.7.

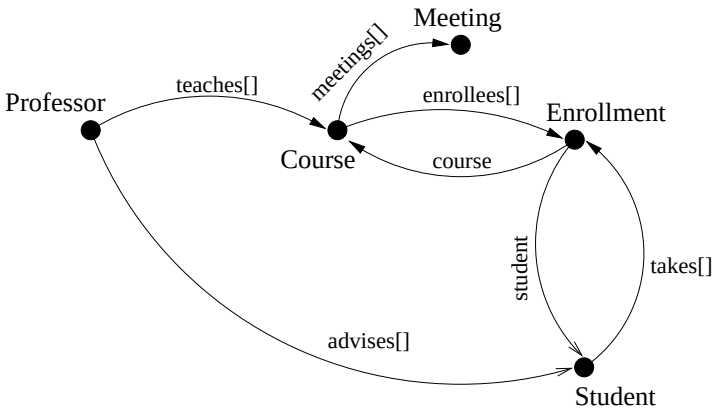

Figure 5.7: The Iom graph produced by combining two annotated itineraries

5.5.1 Adding Class Members From Source and Destination Conditions

Source and destination conditions specified in an itinerary may refer to members of the source and destination classes, respectively. These classes must support such members. The conditions may also refer to members of the visitor class associated with the itinerary, constraining it as well. Here is an example of a leg with source and destination conditions:

```
[Professor,
  teaches[]:Course,
  Professor.name:String==toGradStudents_Visitor.pname:String, //source
  cond
  Course.isGrad():boolean  //dest cond
]
```

This leg requires the Professor class to have a name field of type String, and the Course class to have a boolean function isGrad(). In addition, it requires the toGradStudents_Visitor class to have a pname field of type String. This is shown below in Cb form.

```
Professor =
   <teaches>* Course
   <name> String
.
Course =
   <isGrad()> boolean
.
toGradStudents_Visitor =
   <pname> String
   ...
.
```

5.5.2 Transforming Cb Code to Java Programs

The final step in combining, done by cb2Java, is to generate Java code from the Cb file, in accordance with user preferences. These are some of the preferences that can be set:

- Separate files for each class.
- Default constructor - build constructors that take no arguments, and initialize fields to nulls, zeros, etc.
- Full constructor - build constructors with a formal parameter for each field, used to initialize the fields.
- Default visibility for fields - private, protected, public, none.
- Public getters for each field.
- Public setters for each field.
- Override equals() to do a deep comparison.
- Override clone() to do a deep copy.
- Visit methods, as in: visit(Visitor v) — traverses to each aggregate component of an object calling its visit method. Iterators are used to traverse multi-valued fields. Also generates a skeleton Visitor class with before and after methods for each class in the itinerary list.

Most of the preferences are to support programming style guidelines, as advocated by authors such as Horstmann [53]. The automatic generation of visit methods supports the Visitor Design Pattern [44]. The Cb files can also be used to generate C++ code, with a few more preference options, such as deep destructors, copy constructors, and overloaded assignment operators.

5.5.3 Bottom-Up System Design by View Integration

An important object-oriented design technique, introduced by Jacobson [58], and now part of UML [35], is *use-case analysis* [96]. A use case tracks an interaction between an external user (actor), and the system to be built. Each use case has a name, pre and post conditions, and one or more scenarios (usually a single main scenario, and a number of exceptional ones). A scenario may be pictured using a sequence diagram, showing messaging between domain objects arranged in time order, as in Figure 5.8.

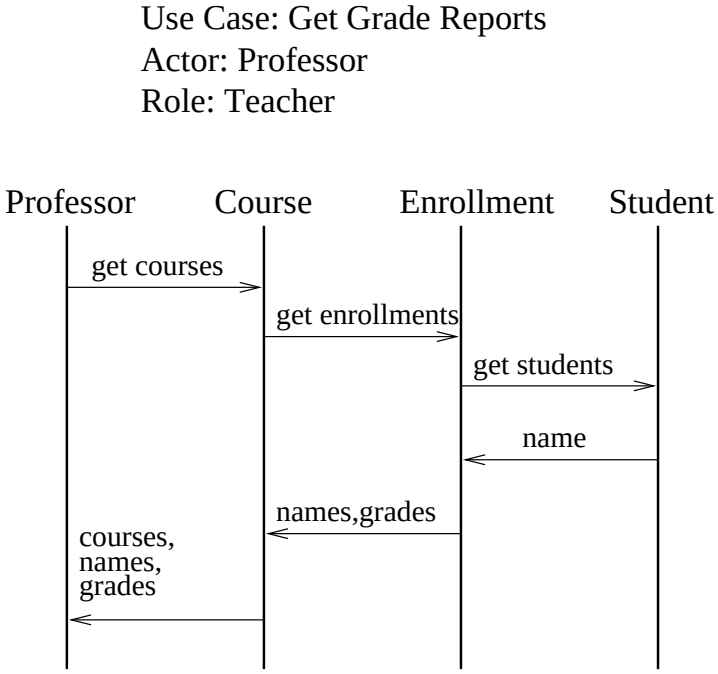

Figure 5.8: A Sequence Diagram For a Use Case

Itineraries can be used to describe the scenarios of a use case, easily capturing the static relationships required to support the messaging shown in an interaction diagram. The analysis stage uncovers a list of use cases that need to be supported in the completed system. The next step is to integrate the use cases with each other, by identifying common classes, members. etc. In a sense, each use case represents a different view of the system as seen by an actor in the exercise of one of her roles. Integrating the use cases amounts to identifying shared structure, as in [111], and shared phenomena as in [57], and making them conform as in schema integration [65]. Once this has been done, the completed system is automatically created from the list of itineraries corresponding to the integrated use cases.

5.6 Using Itineraries to Limit Access

End-users who run applications on a shared system should be limited in their access to objects on a *need to know* basis. There are 3 dimensions to this limitation:

1. Limiting the classes which can be accessed.
2. Limiting the subset of objects in each class's extent which can be accessed.
3. Limiting access to individual members in each class.

These concerns were addressed in [112]. The basic idea is similar to the database mechanism of granting privileges on views, rather than on base tables. The views in this case consist of so-called *secure* itineraries. A secure itinerary differs from a regular itinerary in that shadow classes of the domain classes are visited during a traversal, not the domain classes themselves. These shadow classes contain only methods suitable to the secure itinerary's need to know.

Attaching a secure itinerary requires constructing the shadow classes, and the traversals through them. Shadow classes do not generally contain local state information; this is kept securely in the base classes, and hidden from the secure itinerary. Shadow classes may contain selective accessor and mutator functions, which reference the local state of the associated base classes. Filters to selectively limit access to multi-valued base class fields, may be used. The HAS-A link leading from a shadow class to its base class is kept private, hidden from programmers using the secure itinerary.

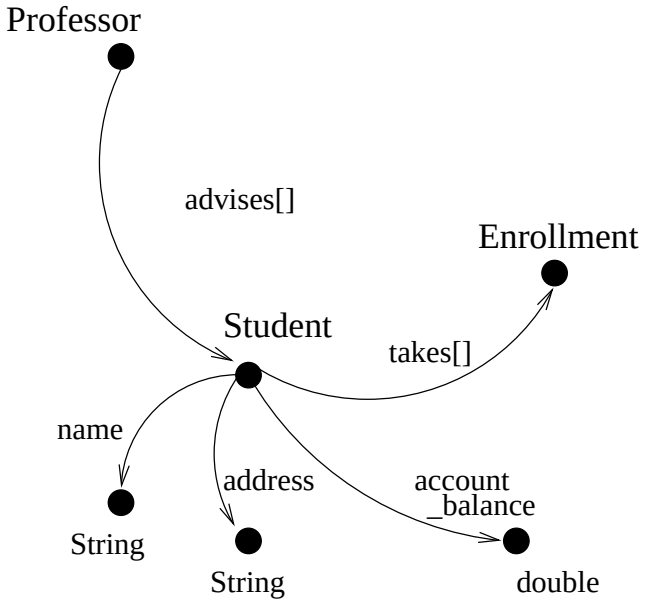

Figure 5.9: An advisor needs to know courses a student is enrolled in, not his account balance

An example should make these concepts clearer. Consider the Iom Graph in Figure 5.9. A professor, acting in her advisor role, needs to know courses a student is enrolled in, not his account balance. This is accomplished by having a Student object construct a *shadow* version of itself, when visited by an advising itinerary. The itinerary specifies the methods to be available in the shadow, as in:

```
ITINERARY toAdvisees
```

```
[Professor, advises, , ] [Student, takes, , ]
@Student{getAddress(), getName()}
```

The shadow object for each class visited is specified by selecting members of the class to be available to the shadow. A shadow designation comes at the end of an itinerary, starts with "@" followed by the class name, and then lists the selected members between curly braces, separated by commas.

When attaching the toAdvisees itinerary above, a new shadow class of Student is defined, with the name toAdvisees_Student. The class definition is:

```
class toAdvisees_Student{
   private Student student;
   public toAdvisees_Student(Student student){this.student = student;}
   public String getName(){return student.getName();}
   public String getAddress(){return student.getAddress();}
}
```

The itinerary's Land method at Student is written to create a toAdvisees_Student object named shadow, on the fly, and to pass it to the itinerary visitor's before and after methods.

```
class Student{
    public boolean toStudents_via_advises_Land(toStudents_Visitor v){
       boolean ret = true;
       toAdvisees_Student shadow = new toAdvisees_Student(this);
       v. before (shadow);
       v. after (shadow);
       return  ret;
   }
}
```

The toStudents_Visitor is limited to calling the getName() and getAddress() methods available at the shadow class, and cannot follow the account_balance link.

Secure itineraries do not, in themselves, limit access to private data. Rather, they are a mechanism that needs to be combined with a system for granting privileges, and enforcing that privileges are not violated at run-time. What secure itineraries do provide is a means of partitioning a large system into smaller chunks on which privileges can be granted. Furthermore, the chunks match up with views of the system, as required by application programs, so that a grant on a single itinerary will usually suffice for such programs.

5.7 Virtual Itineraries

The itineraries spoken of so far are *concrete*, in that they refer directly to classes and links in the underlying schema. We now consider *Virtual Itineraries*, which are more abstract kinds of itineraries, which are then mapped to concrete ones. Continuing the travel industry

metaphor, consider an itinerary which originates in Boston, and includes visits first to London and then to Paris. The itinerary can be pictured as:

$$BOS \rightarrow LON \rightarrow PAR$$

Here BOS is the official designation for Logan Airport in Boston, while LON and PAR are generic designations for airports serving London and Paris respectively. At a further level of refinement, this *virtual* itinerary is mapped to one naming specific airports and flights.

$$BOS \overset{AA-101}{\longrightarrow} JFK \overset{BA-238}{\longrightarrow} LHR \overset{BA-332}{\longrightarrow} ORY$$

Here, LON has been mapped to LHR (London-Heathrow), and PAR to ORY (Paris-Orly), the leg from Boston to London has been telescoped to include a stop in New York's JFK Airport, and the legs have been filled in with specific flight numbers.

A *Virtual Itinerary* is an itinerary, where some of the class names and links may be taken from the underlying schema, while others may be *virtual* in the sense that they will later have to be mapped to actual schema constructs. This is similar to *strategies* in Adaptive Programming [87]. A virtual itinerary can be seen as a constraint which, combined with an actual schema, defines a (possibly empty) set of concrete itineraries. A leg in a virtual itinerary need not map directly to a leg in a concrete itinerary, rather it might map to a path of legs, whose start and end points match the virtual leg.

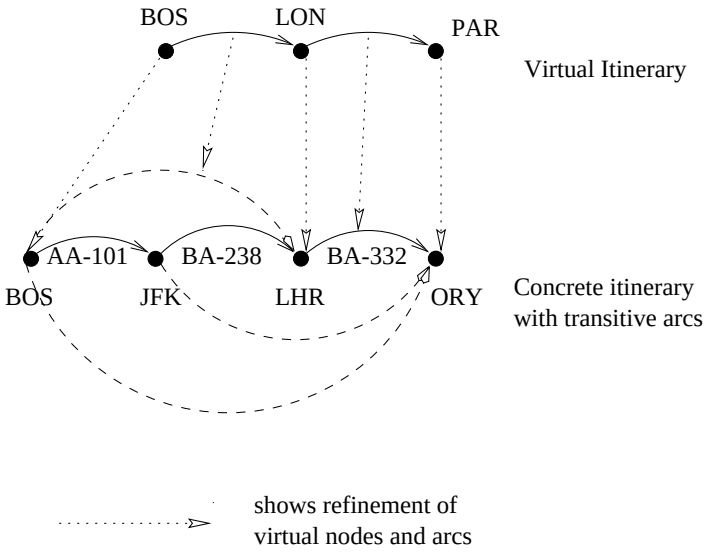

Figure 5.10: A concrete itinerary which refines a virtual itinerary

In the airline example, the concrete itinerary

$$BOS \overset{AA-101}{\longrightarrow} JFK \overset{BA-238}{\longrightarrow} LHR \overset{BA-332}{\longrightarrow} ORY$$

is a refinement of the virtual itinerary

$$BOS \rightarrow LON \rightarrow PAR$$

Roughly speaking, an itinerary I_2 is a refinement of an itinerary I_1 when the IOM graph for I_1, after a mapping of class names, is a subgraph of the transitive closure of that for I_2. This is shown in Figure 5.10.

5.7.1 Specifying Virtual Itineraries

A virtual itinerary is specified similarly to an annotated itinerary, with identifiers for class names, HAS-A names, etc.; however some names are left out in the virtual itinerary, to be filled in by refinement. For example, in the itinerary:

$$[Professor, : Student, ,][Student, : Residence, ,]$$

the names of HAS-A's have been left out, although annotations show their target classes. This virtual itinerary indicates a path that starts at a Professor object, follows an unnamed virtual HAS-A to Student objects, then a virtual HAS-A to Residence objects. Accompanied by an identity name mapping: {Professor$\longrightarrow$Professor, Student$\longrightarrow$Student, Residence$\longrightarrow$Residence}, the itinerary is refined into a collection of paths, from Professor to Residence via Student, and could be useful to a Professor who had to send a mailing to all her students.

5.7.2 Refining Virtual Itineraries and Compatibility With Existing Object Systems

Compatibility of a concrete itinerary with an existing system of Java class definitions was defined in Section 5.4. We now define the notion of a refinement mapping from one itinerary to another. If I is an itinerary, virtual or concrete, let classes(I) be the set of all classes whose names are introduced in I, either as source classes or in an annotation.

An itinerary*refinement* is a function m:$I_1 \longrightarrow I_2$, where I_1 and I_2 are itineraries. It defines a mapping from classes(I_1) to classes(I_2), and a mapping from the legs of I_1 into paths of legs in I_2 as follows:

1. For each class $C \in classes(I_1)$:

$$m(C) \in classes(I_2)$$

2. For each class $C \in classes(I_1)$, each of its member fields or operations as indicated in source and destination conditions is also a member of m(C).

3. For each leg L = [C, h,$cond_s$,$cond_d$] $\in I_1$, there is a path of legs

$$[C_1, h_1, cond_1,][C_2, h_2, ,]...[C_m, h_m, , cond_m]$$

such that m(C) = C_1, m(target(h)) = target(h_m), $cond_1 = cond_s$, and $cond_m = cond_d$.

If m is an itinerary refinement function from I_1 into I_2, we say that I_2 is a refinement of I_1.

Finally, we can define compatibility for virtual itineraries. Given a system S of Java class definitions, a virtual itinerary IV is *compatible* with S if and only if there exists a concrete itinerary IC, such that IC is compatible with S, and IC is a refinement of IV.

5.7.3 Support for Automatic Refinement of Virtual Itineraries

A virtual itinerary IV, together with a mapping m of its names, can be shown to be compatible with an existing system S of Java class definitions if a concrete itinerary IC can be found which is both compatible with S and a refinement of IV. The process of finding such a concrete itinerary can be automated. The refinement is done by using automatic path completions, to map legs in IV into paths of arcs in S. So, if A and B are class names in IV, and there is a leg [A, h, .., ..] in IV where target(h) = B, then the goal is to find a path of arcs,

$$C_1 \stackrel{m(h)}{\longrightarrow} C_2 \longrightarrow ... \longrightarrow C_m$$

in the Iom graph for S, where C_1 = m(A), and C_m = m(B).

There are several approaches to finding such a path:

1. First shortest path — Use breadth-first search from m(A) to m(B). (If HAS-A h is mapped, search from target(m(h)). Take the first path so discovered.
2. All shortest paths — Use breadth-first search from m(A) to m(B) as above. Add all paths that are at the same depth from m(A) as the shortest path. Use the itinerary union (+) operator to add on additional paths.
3. All Paths (broadcast) — Use either depth-first or breadth-first search to discover all paths from m(A) to m(B). This may result in an infinite number of paths if there are cycles in the Iom graph for S.

Typically, a shared system will have cycles; for example, a university system would have paths from Professor to Student to serve Professor needs such as class lists, and also paths from Student to Professor to serve Student needs, such as advisor's office hours. Also, in any system following the ODMG standard [29], inverse links are provided for every association link between classes. Given the prevalence of cycles, the All Paths method is not usually feasible except for applications such as compilers, where the predominant data structure is a tree.

On the other hand, the shortest path methods may fail to uncover important paths between classes. For example, refining the one-leg virtual itinerary [Professor, :Student, .., ..] against the IOM graph in Figure 5.1 will find only the path:

$$Professor \stackrel{advises[]}{\longrightarrow} Student$$

and miss the important path:

$$Professor \stackrel{teaches[]}{\longrightarrow} Course \stackrel{enrollees[]}{\longrightarrow} Enrollment \stackrel{student}{\longrightarrow} Student$$

Both paths are obtained by partially refining the virtual itinerary by hand to:

([Professor, advises[]:Student, .., ..] + [Professor, teaches[]:Student, .., ..])

and then allowing automatic refinement.

Other problems potentially resulting from automatic path completion are the so-called *zigzag* and *shortcut* paths, described in [87]. In the Demeter system, shortcuts and zigzags refer to situations where the propagation graph (the set of generated paths) contains paths which would not be allowed by the propagation directive. In terms of itineraries, a shortcut or zigzag would mean a concrete itinerary, which does not conform to the virtual itinerary it is supposed to refine. For example, suppose there is a virtual itinerary from A to B, which contains a join, say at class X, as in [A,:X,,][X,:B,,]. A shortcut would be a concrete itinerary from A to B, which does not pass through X. However, the rules for refining itineraries would not allow this. Any concrete itinerary which is a refinement of [A,:X,,][X,:B,,] would consist of a path from A to X, followed by a path from X to B. The situation for zigzags is similar. Consider the virtual itinerary

$$([A,:B1,,][B1,:C,,][C,:D1,,][D1,:E,,]+[A,:B2,,][B2,:C,,][C,:D2,,][D2,:E,,])$$

An example of a zigzag would be

$$[A,:B1,,][B1,:C,,][C,:D2,,][D2,:E,,]$$

However, this could not occur according to the rules for refining itineraries.

5.8 Transformations to Itineraries

Itinerary transformations are motivated by the goal of providing automatic maintenance of itineraries in the face of schema change. In Chapter 2, a number of lower level primitives were described to transform existing systems of class definitions. If a list of itineraries has been attached to such a system, the code attached by the itineraries will be transformed as well. However, the original itineraries will now be out of synch with the transformed system, possible referring to names that have been changed, operations which have been delegated, etc. To prevent this, many of the lower level primitives have matching itinerary operations that transform itineraries, so as to make them conform.

Not all CSL primitive operations listed in Figure 2.7 require matching itinerary transformations. For example, ADD CLASS does not affect existing itineraries, and DROP CLASS would not be allowed if an attached itinerary referred to the class. On the other hand, itineraries routinely refer to class names and HAS-A's, and in source and destination conditions to attributes and operations as well. Figure 5.11 shows the primitives which require matching itinerary transformations.

The Rename operations are straight-forward. Remultiply operations require inserting or removing iterators. Delegate and Reclaim are the interesting primitives, requiring in some cases that a leg be telescoped or contracted.

Consider the itinerary: [Professor, advisees, ,]. Following the transformation:

DELEGATE FIELD advisees SOURCE Professor USING HAS-A advisor

	Rename	Remultiply	Delegate	Reclaim
Class	X			
Attribute	X	X	X	X
Operation	X		X	X
HAS-A	X	X	X	X

Figure 5.11: Itinerary Transformations

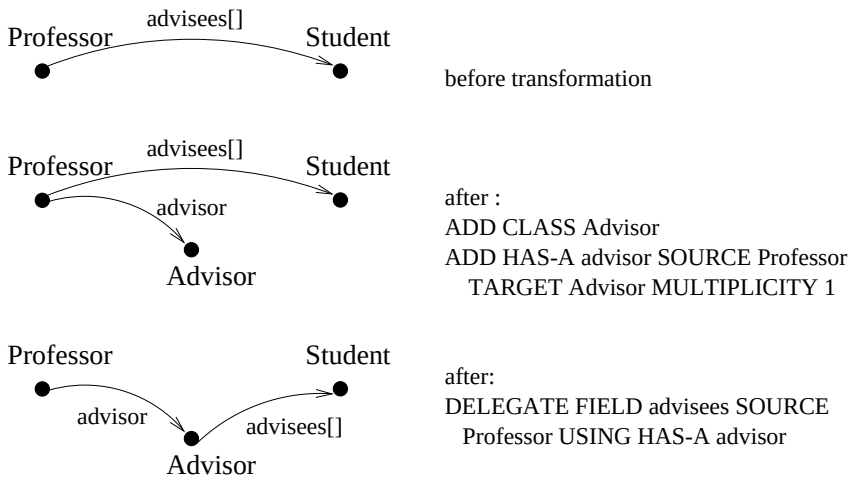

Figure 5.12: Transforming an itinerary after a delegate operation

as indicated in Figure 5.12, this 1-leg itinerary must be telescoped to the 2-leg itinerary: [Professor, advisor, ,] [Advisor, advisees, ,].

Itinerary operations are applied as follows:

1. The itinerary list is parsed into a data structure.
2. Itinerary updates are represented as concrete transformation visitors.
3. Updates are applied to the data structure.
4. The updated data structure is printed out as a revised itinerary list.

Higher level primitives such as merge, split, aggregate and decompose are also supported, in that the low level primitives which they spawn have primitives that transform itineraries.

5.9 Work Related to Itineraries

5.9.1 Traversal Specifications

There has been recent work by Ovlinger and Wand [83] on building a language to specify traversals of object systems. This language is similar to the itinerary language in that it

allows for an explicit definition of a traversal to be carried out on a class diagram which is *consistent* with it. There are provisions for passing parameters and obtaining return values. Also, visitor methods can be explicitly called. There is no provision for checking conditions before and after traversing a link, nor is there a way to combine traversals using joining and unioning. On the other hand, traversals can be nested. Nesting is particularly useful when a single class plays multiple roles, such as a Person playing variously the roles of sibling and spouse. See Figure 5.4.

Whereas itineraries attach code to existing Java classes, Ovlinger's traversal specifications concentrate the code into a single traversal class. This class necessarily encapsulates schema information matching the traversal, and is sensitive to schema change. Of course it can be regenerated. It should be noted that itineraries can be automatically maintained in event of schema change by applying the same CSL transformations used to modify the schema.

5.9.2 Aspectual Components

Aspectual Components [68] is a recently developed means for programming class collaborations. It employs a *participant graph* which is similar to a virtual itinerary, in that the nodes represent virtual classes called participants, which must later be bound to concrete ones, and the links represent structural constraints on the concrete classes similar to IS-A and HAS-A IOM links. Aspectual components interface with each other at *abstract join points*, similar to the pins in Pinned Views [111], or the intersections of several itineraries at a class. A participant may also reimplement a feature such as a method by replacing its body with another.

5.9.3 UML OCL

Itineraries, including virtual ones, act as a constraint on the possible class graphs which are compatible with them. It is not surprising that there is a similarity between itineraries and the UML Object Constraint Language [33] (OCL).

Like itineraries, OCL supports navigation along association links, which are either single or multiple valued. A binary UML association amounts in effect to two Iom HAS-A links, one in each direction. OCL uses the role name at the target end of the association when indicating navigation. In the absence of a role name, it generates one from the name of the target type, by making the first letter lower case. Navigation through either a single-valued or multiple-valued association can be assumed to result in a set of objects. OCL also has a feature called *qualified associations*, which selects from this result, according to the values of specified attributes. The more powerful *destination condition* of itineraries can accomplish the same thing. OCL allows the specification of *select* operations on collections, using arbitrary boolean expressions. Selecting from the set of objects obtained by navigating a link is equivalent to applying an itinerary destination condition.

5.9.4 Database Views

There is a connection between itineraries and recent research on bringing ANSI/SPARC [108, 40] type view mechanisms into object database systems. Both itineraries and views provide mechanisms for presenting smaller, simplified versions of a large object system for use by selected applications and end-users. It is helpful at this point to examine the state of the art in research on object-oriented views.

A database view can be designed for an individual user or host program. A view gives a simplified and limited access to the database, and is useful in providing security and in mitigating the effects of schema change on users. The set of defined views forms the external level in the three-level ANSI/SPARC architecture [108, 40]. The levels are:

1. *internal* level — Describes the physical storage structure of the database.
2. *conceptual* level — A global description of the structure of the database in terms of entity types (base tables), data types, relationships and constraints.
3. *external* level — The set of defined views which form the interface for the database users.

Relational Database Views

A relational database view is defined by storing an SQL SELECT statement in a data dictionary table of views. The view definition is, in effect a mapping from the conceptual schema to the view schema. The mapping may require joining tuples from several different tables, projecting and/or renaming attributes, and restricting the tuples in the view to those which satisfy a predicate. When an application program or ad-hoc user queries a view, the view is *materialized* by executing the SELECT query; the user (or program) can select from the view as if it was a base table, and in certain cases [1] can update from it as well.

Logical data independence refers to the ability to change the conceptual level while still supporting existing views defined at the external level. This is done by replacing old view queries, where needed, with new ones which are equivalent from the client's point of view. Provided the view query can be updated, an application program which accesses the database through a view does not need to be altered or even recompiled.

Views can be used to restrict access to the total data so that individual users see only what is pertinent to their needs [93, 100, 27]. Users are granted privileges such as SELECT, DELETE, or UPDATE on one or more view tables and are limited to what these privileges allow.

Views are also useful for creating the global conceptual schema in the first place as part of bottom-up design [78, 65, 40]. A schema consisting of several tables and relationships between them is identified to serve the needs of each user group. These view schemas are then analyzed to identify correspondences and conflicts between them. They are then

[1] for example, if the view is derived from a single table and includes its primary key

made to conform to each other by means of renaming and restructuring. Once the view schemas conform, they may then be merged together to create the global conceptual schema. Some further restructuring and/or normalization may be done at this point. Finally, the view definitions are written and stored in the data dictionary so that the views can be materialized on demand.

Object-Oriented Database View Research

This section surveys some of the research that has been done in object-oriented views [63, 1, 64, 92, 98, 17]. One of the motivations for developing object-oriented databases was to reduce the so-called *impedance mismatch* that occurs between application programming languages such as Smalltalk and C/C++, which can handle richly structured data, and relational databases and their query languages such as SQL, which can handle only flattened tables [30, 74]. Using an object-oriented database with an object-oriented language effectively eliminates the impedance mismatch. The host language can handle data storage and retrieval by itself, without the need for a query language. However, there is still a need for a view mechanism to facilitate schema evolution, provide for authorization control, and aid in program and schema development.

The basis for views in relational databases is the use of a separate, declarative query language for all data access. It appears at this point that two families of query languages will be of general use for object-oriented databases [74]. The extended relational databases such as POSTGRES, Montage, Starburst and UniSQL [30] use extensions of the relational query languages QUEL (POSTGRES) or SQL (the others). Other object-oriented databases, including ONTOS, ObjectStore, O_2, Gemstone, Itasca, Objectivity/DB, Versant and POET, are expected to support Object Query Language (OQL), a standard introduced by the industry consortium *Object Database Management Group* (ODMG) [28, 30]. OQL closely resembles SQL-92 query syntax, and the ODMG is expected [110] to further conform OQL to be the read-only query subset of SQL3.

UniSQL/X [63, 62] is an example of the extended relational approach. Its view mechanism extends the semantics of relational views by considering methods, object identity, inheritance and aggregation. A UniSQL/X view is defined by specifying a view name, a list of attributes (and their domains), a list of methods, a list of superclasses, and a query specification. In this example a view is created from an existing base class, Employee:

```
Create View EmpPay
(SSN, Total-Pay: Real, Department) As
Select SSN, Salary + Commission, Department
From Employee;
```

Since the view is based on a single stored class, an object in it can be materialized using the Object ID(OID) of the base object. This allows updates from the view to be propagated back to the base object. As in the relational case, Total-Pay is not updatable since it is a derived attribute.

OQL [29], while it doesn't support views, does include a *define* statement which associates a name with a query expression for later use. An example from the proposed OQL standard:

```
define jones as select distinct x from Students x
where x.name = "Jones";
```

This is similar to naming and storing views in relational databases.

Bertino [17] proposes a view mechanism that goes well beyond the relational concept of view by allowing views to contain additional properties and methods beyond those available in the base classes. An example:

create-view ExtForeignMachine
view-query x; x/Machine; x.manufacturer.location.country ¬ ='Italy'
additionalproperties (fprice: numeric)

The view contains an additional attribute, fprice, which requires storage beyond that used by the base classes.

Bertino's view mechanisms accommodate certain kinds of schema change including adding and dropping properties, changing an attribute's domain, and changing method implementations. These changes are largely made possible by the ability to store additional properties and methods in the view. However such views blur the distinction between view and base classes, in seeming violation of the 3-level ANSI/SPARC architecture [108, 40].

Multiview [64, 92, 98] is a system developed at the University of Michigan by a team led by Elke A. Rundensteiner. It provides for views into an object-oriented database using object-slicing techniques. Each entity object is represented by a single conceptual object and a set of implementation objects. It is the implementation objects which carry the local state and support behavior. The implementation objects are themselves arranged into an inheritance hierarchy. A view which includes an implementation object would include the properties and behavior of that object as well as its ancestor implementation objects.

A grammar approach to view definition is offered by Ken Baclawski [9]. There is a distinguished entity set (class) which serves as both the focal point of the view and the start symbol of a set of rewrite rules in a context-free grammar. A view consists of the focal class together with classes reachable from it by applying the rules. A sentence which can be parsed by the rules represents a view instance. Taken together, such view instances may contain redundancies, as when a class on the left hand side of a rule is in a many:one relationship with a class on the right hand side. Baclawski uses the term *panorama* to describe a maximal non-redundant view.

Role of Views in Schema Evolution

Allowing direct access to data by application programs makes them extremely vulnerable to schema change precisely because they contain too much schema information. Nevertheless, Cattell asserts [30] that"...encapsulation provides data independence through the

implementation of methods, allowing the private portion of an object to be changed without affecting programs that use that object type." For example, a program, instead of accessing an attribute directly, would use publicly available get and set methods for that attribute. However, when there have been numerous schema changes it is hard to see how the same get and set methods could serve multiple programs created at different times. There is still a need either for views set up for individual programs, or for the programs themselves to change to keep current with the schema.

UniSQL/X views remain valid following certain types of schema change. For example, adding a new attribute or method would not affect the view unless it had been defined using the wild card * in its Select statement. Dropping an attribute or method would invalidate those views which included it. Renaming attributes and methods would require redefining the views which use them. Changes to the inheritance hierarchy have effects similar to adding or dropping attributes and methods. The philosophy of UniSQL/X is to issue warnings to users of potentially invalidated views, rather than to attempt automatically changing the view definitions themselves.

A schema version mechanism, Transparent Schema Evolution (TSE) is built on top of MultiView. The goal of TSE is to enable new, updatable views to be developed without affecting programs using existing views. TSE views are capacity-augmenting in that new stored attributes and methods may be defined for a view class. TSE requires the underlying object system to support multiple classification and dynamic reclassification of object representations. These facilities are available in MultiView, but are not generally available in commercial databases.

Limitations of Views

Although views remain an active area for academic research they appear to be fundamentally limited in their ability to provide support for schema evolution.

- View schemas are inhospitable to navigating programs. The DBMS must take control each time a reference is followed in order to ensure that it is properly dereferenced within the view.
- Programs with an archaic view of the world may be unsound. For example, they may attempt to construct objects in classes that no longer exist, or miss out on important data by failing to traverse newly added pathways.
- Update capability from a view is limited.
- Capacity augmenting views as in [17, 64, 92, 98] in effect privatize what was formerly shared data by limiting its access to within the view.

There has been some work on building views which present networks for schema navigation. For example, Bertino [17, 18] proposes a view mechanism consisting of *views*, which are virtual classes, and *view schemas*, which consist of a collection of views, closed under aggregation and inheritance. Bertino's views are either *object-preserving*, in that all view

objects are extracted from base classes, or *object-generating*, meaning that the views may contain instances that appear in no base class. As with itineraries, the over-all goal is to provide different applications with their own customized view of the data. Bertino has also done work towards using views for authorization and access control [20, 19].

Minsky [76] proposes supporting views by means of surrogate objects, similar to Section 5.6. Multiview [64, 92, 98] spreads both local state and functionality over several *implementation objects* centered on one *conceptual object*.

5.10 Summary

This chapter has presented the itinerary language for capturing navigational information needed by programs using shared object systems. An itinerary is an intuitive concept using the familiar metaphor of airline travel. End-user applications can be built by employing the visitor design pattern together with itineraries.

Itineraries extend the power of the STP transformations described in Chapter 2, by providing support for adding on new applications to an existing system.

Itineraries can also be combined. By combining all the itineraries available to a specific class of end-users, a kind of sub-schema view is generated. This view shows all classes needed by these end-users, however only some of the members of each class are included. This is in accordance with what these end-users *need to know*. Such views can form the basis for introducing security and privacy into shared systems. In Section 5.9.4 itinerary views were compared to related work in object-oriented database views.

Itineraries are easily specified, and can be attached to an existing object system. They can be transformed using the same CSL commands as transform object systems. It is also possible to specify *virtual* itineraries, and to refine them to actual ones. Finally, a prototype system is available which demonstrates the practicality of these ideas.

Chapter 6

Conclusion

6.1 Summary of Contributions

The chief contributions of this work lie in these areas:

1. Development of a modeling language, IOM, including its graphical representation, IOM Graph, which exposes class design in terms of labeled nodes and labeled arrows, exclusively.

2. Development of a Change Specification Language (CSL), which expresses desired schema changes as English commands which refer to constructs in the IOM model.

3. A careful examination of a large variety of schema changes in terms of pre and post conditions.

4. Proof of feasibility of the automatic application of changes expressed in CSL by constructing a working prototype, Schema Transformation Processor (STP), and testing it by transforming a number of Java programs.

5. Development of the Itinerary Language to express the navigational needs of programs which require the cooperation of a number of participating classes to do their job. Itineraries complement CSL transformations by providing support for adding on new applications to an existing system, without changing the schema. The new behavior is encapsulated in a visitor object, with navigational support provided by attaching its itinerary.

6. The feasibility of itineraries has also been shown by prototype programs which attach itineraries to existing object systems, build new systems from a set of itineraries and apply CSL transformations to existing itineraries.

6.2 Limitations of Approach

As it stands, the system of transformations outlined in Chapters 2 and 3 suffers from several limitations.

1. The problem cited in Chapter 4 of Java collection classes, where the underlying type of the objects stored is kept hidden. This causes a programming problem, in that objects retrieved from such collections must frequently be downcast to more usable types, and also a problem for reverse engineers, who must attempt to infer the underlying type in order to show it in design models.

2. Changes cannot be applied to imported classes, even though such classes can be modelled using the extended IOM.

6.3 Improving the Prototype Programs

As part of this research, a suite of prototype programs has been developed to demonstrate *proof of concept* for the majority of the ideas presented here. The transformation prototypes have been tested on a variety of programs, including samples from the Sun Java distribution kit. While generally successful with these programs, it is hard to measure the efficiency of the prototype, and few claims are made for its robustness. Prototypes have been built for itineraries, which attach and transform them. The generation of Java source code from a list of itineraries has also been prototyped.

The prototypes make extensive use of powerful tools, such as Sun's JavaCC compiler compiler, and the Visitor Design pattern. Also, the use of a working version of the IOM object model has proven to be invaluable. However, since construction began on the prototype programs, new tools have emerged. In particular, Java's enhanced reflection capability, updated versions of JavaCC, and the accompanying tool *Java Tree Builder* (JTB) [86] should prove useful in rebuilding the prototypes to more demanding specifications. The prototypes do have a Graphical User Interface (GUI) which features a diagram editor for IOM Graphs. This should be enhanced, since visual programming methods are particularly suitable for describing both transformations and itineraries.

Experience with refactoring tools [95] has shown that users often experiment with a transformation, then decide to retract it. Consequently, an *undo* facility is vital for these applications. The current prototype lacks an undo facility. This means a user who wishes to experiment must either keep a back copy of the system, or be knowledgeable enough to know how to apply reverse transformations.

6.4 Future Directions

In the course of working on this research, it became apparent that there are several areas which merit further investigation.

1. Use of more powerful mathematical tools, such as category theory.

2. Binding design models and programming languages to each other in a consistent way, in order to foster automatic code generation from designs, on the one hand, and automatic design extractions from code on the other.

3. Extending encapsulation in object-oriented systems into views that encompass multiple cooperating classes. The research on Chapter 5 is a beginning on this, but more needs to be done.

4. Building a sound security protocol for object-oriented databases, so as to limit an individual's access to the system on a *need to know* basis.

5. Using itineraries to trim down the size of objects before shipping them over data communications lines.

6. Making use of the Java serialization standard [42], so as to be able to read back a stored object following the application of CSL changes.

6.5 Summary

One contention of this thesis has been that it is possible to automatically update a set of source programs, which together generate a shared object system, by giving a list of commands which specify changes in terms of object model rather than programming language constructs. To do this a simplified object model along with an arc-oriented graphical representation were introduced. There was an extensive examination of the preconditions necessary to assure continued type soundness of the system, as well as the additional preconditions needed to assure behavior preservation.

These ideas were borne out in practice by the construction of a prototype program which has been tested by applying lists of transformations to Java programs supplied as samples in Sun's toolkit, and in introductory textbooks on Java.

A second contention of this thesis has been that it is possible and worthwhile to define view-like mechanisms called itineraries, which capture the navigational needs of individual programs, or individual classes of end-users, in a shared object system. It was shown how to define and to attach itineraries to an existing system, how to write programs by combining itineraries and visitors, how to transform itineraries using the same commands as used to transform programs, and also how to combine itineraries to build the schema for a shared system. These ideas have also been shown to be feasible by building prototype programs.

Of course, there remains much to be done. Perhaps the change specification language described here, or something close to it, can be incorporated as a standard by the Object Management Group (OMG). Research into itineraries, as in other view-like mechanisms for object systems, is at an early stage. Itineraries may be especially helpful in efforts to bring some security and limits on access into object databases. Hopefully this thesis will serve as a start for these projects.

Bibliography

[1] Serge Abiteboul and Anthony Bonner. Objects and Views. In *Proc. Intl. Conf. on Management of Data*, pages 238–247. ACM SIGMOD, May 1991.

[2] J. R. Abrial. *Data Semantics*, pages 1–60. North Holland Pub. Co., Amsterdam, 1974.

[3] Ole Agesen, Stephen Freund, and John C. Mitchell. Adding type parameterization to java. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, Atlanta, GA, 1997. ACM.

[4] Suad Alagic. The ODMG object model: Does it make sense? In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 253–270, Atlanta, GA, 1997. ACM.

[5] Andrew W. Appel. *Modern Compiler Implementation in Java.* Cambridge University Press, 1998.

[6] Andrew W. Appell. SSA is functional programming. *SIGPLAN Notices*, 33(4):17–20, April 1998.

[7] Ken Arnold and James Gosling. *The Java Programming Language.* Addison-Wesley, 1996.

[8] Charles W. Bachman. The programmer as navigator. *Communications of the ACM*, 16(1):653–658, November 1973.

[9] Kenneth Baclawski. Panoramas and grammars: A new view of data models. Technical report, Northeastern University, April 1993.

[10] Francois Bancilhon, C. Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database System: The Story of O2.* Morgan-Kaufmann, 1992.

[11] 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.

[12] Michael Barr and Charles Wells. *Category Theory For Computing Science.* Prentice Hall, 1990.

[13] K. Beck and W. Cunningham. A laboratory for teaching object-oriented thinking. In *Proceedings OOPSLA '89, ACM SIGPLAN Notices*, October 1989. Published as Proceedings OOPSLA '89, ACM SIGPLAN Notices, volume 24, number 10.

[14] 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).

[15] Paul Bergstein. *Syntax-preserving evolution of class structures.* PhD thesis, Northeastern University, 1994.

[16] Paul L. Bergstein and Walter L. Hürsch. Maintaining behavioral consistency during schema evolution. In S. Nishio and A. Yonezawa, editors, *International Symposium on Object Technologies for Advanced Software*, pages 176–193, Kanazawa, Japan, November 1993. JSSST, Springer Verlag, Lecture Notes in Computer Science.

[17] Elisa Bertino. A View Mechanism for Object-Oriented Databases. In *Advances in DB-Technology, Proc. Intl. Conf. on Extending Database Technology (EDBT)*, number 580 in Lecture Notes in Computer Science, pages 136–151, Vienna, Austria, March 1992. Springer.

[18] Elisa Bertino and Giovanna Guerrini. Viewpoints in object database systems. In *Viewpoints 96: An International Workshop on Multiple Perspectives in Software Development*, San Francisco, CA, oct 1996. ACM Press. http://www.cs.city.uk/homes/gespan/vptoc.html.

[19] Elisa Bertino, Sushil Jajodia, and Pierangela Samarati. Supporting multiple access control policies in database systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, 1996. IEEE.

[20] Elisa Bertino, Fabio Origgi, and Pierangela Samarati. A new authorization model for object-oriented databases. In *Proceedings of the IFIP WG11.3 Working Conference on Database Security*, Amsterdam, The Netherlands, 1994. Elsevier.

[21] Grady Booch, James Rumbaugh, and Ivar Jacobson. The unified modeling language for object-oriented development. Version 0.9 Addendum, July 1996.

[22] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the java programming language. www.math.luc.edu/pizza/gj/Documents/, 1998.

[23] Philippe Breche. Advanced primitives for changing schemas of object databases. In *Proc. of the CAiSE'96,Heraklion, Cr/'ete*. Springer Verlag, May 1996.

[24] Philippe Breche, Fabrizio Ferrandina, and Martin Kuklok. Simulation of schema change using views. In *Proceedings of 6th International Conference on Database and Expert Systems Applications (DEXA '95), London*, September 1995.

[25] Frederick P. Brooks. No silver bullet, essence and accidents of software engineering. *IEEE Computer Magazine*, pages 10–19, April 1987.

[26] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, 1997.

[27] Silvana Castano, Maria Grazia Fugini, Giancarlo Martella, and Pierangela Samarati. *Database Security*. Addison-Wesley, 1995.

[28] R. G. G. Cattell, editor. *The Object Database Standard ODMG - 93*. Morgan Kaufmann, 1994.

[29] R. G. G. Cattell, editor. *The Object Database Standard ODMG - 93*. Morgan Kaufmann, 1996.

[30] R.G.G. Cattell. *Object Data Management*. Addison Wesley, Reading, MA, 1994. Revised Edition.

[31] P. Chen. The entity-relationship model – toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[32] Charles Consel. Program adaptation based on program transformation. *SIGPLAN Notices*, 32(1):69–72, January 1997.

[33] Rational Corporation. Object Constraint Language Specification. http//www.rational.com/uml/resources/documentation/index.jtmpl, Sept 1997.

[34] Rational Corporation. UML Notation Guide. http//www.rational.com/uml/resources/documentation/index.jtmpl, Sept 1997.

[35] Rational Corporation. UML Semantics. http//www.rational.com/uml/resources/documentation/index.jtmpl, Sept 1997.

[36] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.

[37] William S. Davis. *Systems Analysis and Design*. Addison-Wesley, 1983.

[38] T. DeMarco. *Structured Analysis and System Specification*. Yourdan Press, 1978.

[39] Dave Dyer. Java decompilers compared. *Java World*, July 1997.

[40] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Benjamin Cummings, 1989.

[41] Fabrizio Ferrandina, Thorsten Meyer, and Roberto Zicari. Schema and database evolution in the O2 object database system. In *Proceedings of the 21th International Conference on Very Large Databases, Zurich, Switzerland (VLDB '95)*, September 1995.

[42] David Flanagan. *Java in a Nutshell*. O'Reilly, Sebastapol, CA, 1997.

[43] Ira R. Forman, Michael H. Conner, Scott H. Danford, and Larry K. Raper. Release-to-release binary compatibility in som. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, pages 426–438, Austin, TX, 1995. ACM.

[44] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[45] James Gosling, Bill Joy, and Guy L. Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[46] Marc Gyssens, Jan Paradaens, Jan Van den Bussche, and Dirk Van Gucht. A graph-oriented object database model. In Hector Garcia-Molina and H.V. Jagadish, editors, *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, Atlantic City, 1990. ACM Press.

[47] Brent Hailpern and Harold Ossher. Extending Objects to Support Multiple Interfaces and Access Control. *IEEE Transactions on Software Engineering*, 16(11):1247–1257, November 1990.

[48] Chris Hankin, Hanne Riis Nielson, and Jens Palsberg. Strategic Directions for Research on Programming Languages. *ACM Computer Surveys*, 28(4), December 1996.

[49] Cole Harrison. AQL: An Adaptive Query Language. Technical Report NU-CCS-94-19, Northeastern University, October 1994. Master Thesis.

[50] 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.

[51] Ian M. Holland. Specifying reusable components using contracts. In O. Lehrmann Madsen, editor, *Proceedings ECOOP '92*, LNCS 615, pages 287–308, Utrecht, The Netherlands, June 1992. Springer-Verlag.

[52] Ian M. Holland. *The design and representation of object-oriented components*. PhD thesis, Northeastern University, 1993.

[53] Cay S. Horstmann. *Practical Object-Oriented Development in C++ and Java*. Wiley, 1997.

[54] Walter Hürsch. *Maintaining Behavior and Consistency of Object-Oriented Systems during Evolution*. PhD thesis, Northeastern University, 1995. http://www.ccs.neu.edu/home/lieber/theses-index.html.

[55] Walter L. Hürsch and Linda M. Seiter. Automating the evolution of object-oriented systems. *The American Programmer*, 4(10):46–56, July 1995.

[56] Daniel Jackson and Allison Waingold. Lightweight extraction of object models from bytecode. In *International Conference on Software Engineering*, Los Angeles, CA, May 1999. http://sdg.lcs.mit.edu/womble/.

[57] Michael Jackson. Connecting viewpoints by shared phenomena. In *Viewpoints 96: An International Workshop on Multiple Perspectives in Software Development*, San Francisco, CA, oct 1996. ACM Press. http://www.cs.city.uk/homes/gespan/vptoc.html.

[58] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering – A Use Case Driven Approach.* Addison-Wesley/ACM Press, Reading, Mass., 1992.

[59] Ralph E. Johnson and William F. Opdyke. Refactoring and aggregation. In *Object Technologies for Advanced Software, First JSSST International Symposium*, volume 742 of *Lecture Notes in Computer Science*, pages 264–278. Springer-Verlag, November 1993.

[60] Mathew Kaplan, Harold Ossher, William Harrison, and Vincent Kruskal. Subject-oriented design and the watson subject compiler. http://www.research.ibm.com/sop/papers/position96.htm, 1996.

[61] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. Technical Report SPL97-008, XEROX/PARC, February 1997.

[62] Michael Kifer, Won Kim, and Yehoshua Sagiv. Querying object-oriented databases. In Michael Stonebraker, editor, *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 393–402, San Diego, California, 1992. ACM Press.

[63] Won Kim and William Kelley. *On View Support in Object-Oriented Database Systems*, chapter 6, pages 108–129. Addison-Wesley, 1995.

[64] H. A. Kuno and E. A. Rundensteiner. Materialized object-oriented views in multiview. 1994. Technical report, University of Michigan.

[65] J. A. Larson, S. B. Navathe, and R. Elmasri. Attribute equivalence and its use in schema integration. *IEEE Transactions on Software Engineering*, 15(2), April 1989.

[66] Georg Lausen and Gottfried Vossen. *Models and Languages of Object-Oriented Databases.* Addison-Wesley, Reading, Mass., 1998.

[67] Sven-Eric Lautemann. A propagation mechanism for populated schema versions. In *13th Int'l Conference on Data Engineering*, March 1997.

[68] Karl Lieberherr, David Lorenz, and Mira Mezini. Programming with Aspectual Components. Technical Report NU-CCS-99-01, College of Computer Science, Northeastern University, Boston, MA, March 1999.

[69] Karl J. Lieberherr. *The Art of Growing Adaptive Object-Oriented Software.* PWS Publishing Company, Boston, 1995.

[70] Karl J. Lieberherr, Ian Holland, and Arthur J. Riel. Object-oriented programming: An objective sense of style. In *Object-Oriented Programming Systems, Languages and Applications Conference, in Special Issue of SIGPLAN Notices*, number 11, pages

323–334, San Diego, CA., September 1988. A short version of this paper appears in IEEE Computer, June 88, Open Channel section, pages 78-79.

[71] Karl J. Lieberherr, Ignacio Silva-Lepe, and Cun Xiao. Adaptive object-oriented programming using graph-based customization. *Communications of the ACM*, 37(5):94–101, May 1994.

[72] 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.

[73] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997.

[74] Mary E. S. Loomis. *Object Databases: The Essentials*. Addison-Wesley, Reading, Mass., 1994.

[75] Mira Mezini and Karl Lieberherr. Adaptive plug-and-play components for evolutionary software development. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, volume 33, pages 97–116, October 1998.

[76] Naftaly H. Minsky and Partha Pal. Providing multiple views for objects by means of surrogates. In *Viewpoints 96: An International Workshop on Multiple Perspectives in Software Development*, San Francisco, CA, oct 1996. ACM Press. http://www.cs.city.uk/homes/gespan/vptoc.html.

[77] Andrew C. Myers, Joseph A. Bank, and Barbara Liskov. Parameterized types for java. In *ACM Symposium on Principles of Programming Languages*, january 1997.

[78] Shamkant Navathe, Ramez Elmasri, and James Larson. Integrating user views in database design. *Computer*, pages 50 – 62, january 1986.

[79] Erik Odberg. MultiPerspectives: The Classification Dimension of Schema Modification Management for Object-Oriented Databases. In *TOOLS USA '94 (Technology of Object-Oriented Languages and Systems), Santa Barbara, California, USA*, August 1994.

[80] Martin Odersky and Philip Wadler. Pizza into java: Translating theory into practice. In *ACM Symposium on Principles of Programming Languages*, january 1997.

[81] William F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois, 1992. ftp://st.cs.uiuc.edu/pub/papers/refactoring/Opdyke-thesis.ps.Z.

[82] H. Ossher, M. Kaplan, A. Katz, W. Harrison, and V. Kruskal. Specifying subject-oriented composition. *Theory and Practice of Object Systems*, 2(3), 1996.

[83] Johan Ovlinger and Mitchell Wand. A Language for Specifying Traversals of Object Structures. Technical Report NU-CCS-98-??, College of Computer Science, Northeastern University, Boston, MA, November 1998.

[84] Jens Palsberg. Class-graph Inference for Adaptive Programs. *Theory and Practice of Object Systems*, 3(2), April 1997.

[85] Jens Palsberg and Michael I. Schwartzbach. *Object-Oriented Type Systems.* John Wiley & Sons, 1993.

[86] Jens Palsberg and Kevin Tao. JTB - Java Tree Builder Documentation. http://www.cs.purdue.edu/homes/taokr/jtb/docs.html, 1999.

[87] Jens Palsberg, Cun Xiao, and Karl Lieberherr. Efficient implementation of adaptive software. *ACM Transactions on Programming Languages and Systems*, 17(2):264–292, March 1995.

[88] Alberto Pettorossi and Maurizio Proietti. Future directions in program transformation. *SIGPLAN Notices*, 32(1):99–102, January 1997.

[89] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* The MIT Press, 1991.

[90] George Polya. *How to solve it.* Princeton University Press, 1949.

[91] Terrance W. Pratt and Marvin V. Zelkowitz. *Programming Languages - Design and Implementation.* Prentice Hall, 3 edition, 1995.

[92] Y. G. Ra, H. A. Kuno, and E. A. Rundensteiner. A flexible object-oriented database model and implementation for capacity-augmenting views. Technical report, University of Michigan, 1994. Technical report, University of Michigan.

[93] Fausto Rabitti, Elisa Bertino, Won Kim, and Darrell Woelk. A model of authorization for next-generation database systems. *ACM Transactions on Database Systems*, 16(1):88–131, March 1991.

[94] Dirk Riehle and Thomas Gross. Role Model Based Framework Design and Integration. In *Proceedings OOPSLA '98, ACM SIGPLAN Notices*, volume 33, pages 117–133, October 1998.

[95] Don Roberts, John Brant, and Ralph Johnson. A refactoring tool for smalltalk. *Theory and Practice of Object Systems*, 1996.

[96] James Rumbaugh. Getting started–using use cases to capture design requirements. *Journal of Object-Oriented Programming*, pages 8–12,23, Sep 1994.

[97] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design.* Prentice Hall, New Jersey, 1991.

[98] Elke A. Rundensteiner. MultiView: A Methodology for Supporting Multiple Views in Object-Oriented Databases. In *Proc. 18th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 187–198, Vancouver, Canada, 1992. ACM SIGMOD.

[99] Stuart Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach.* Prentice Hall, 1995.

[100] Gunter Saake and Ralf Jungclaus. Views and Formal Implementation in a Three-level Schema Architecture For Dynamic Objects. In P. M. D. Gray and R. J. Lucas, editors, *Advances in Databases, Proc. 10th British National Conf. on Databases (BNCOD)*, pages 78–95, Aberdeen, Scotland, july 1992. Springer.

[101] F. Saltor, M. G. Castellanos, and M. Garcia-Solaco. Overcoming schematic discrepancies in interoperable databases. In E. J. Neuhold D. K. Hsiao and R. Sacks-Davis, editors, *Interoperable Database Systems*. North-Holland, 1993.

[102] Sriram Sankar. The java compiler compiler. available from Sun Microsystems.

[103] Sriram Sankar. Java 1.02 grammar. http//www.cs.ucla.edu/ dongwon/JavaCC/java/Java1.02.jj, June 1996.

[104] Stephen R Schach. *Software Engineering with Java*. Irwin, 1997.

[105] Charles Simonyi. Intentional Programming - Innovation in the Legacy Age. In *IFIP WG 2.1*, http://www.advtech.microsoft.com/research/ipi/ifipwg/ifipwg.htm, 1996. Springer Verlag.

[106] Dag Sjøberg. Quantifying Schema Evolution. *Information and Software Technology*, 35(1):35–44, January 1993.

[107] Lance Tokuda and Don Batory. Automated Software Evolution via Design Pattern Transformations. In *Proceedings of the 3rd International Symposium on Applied Corporate Computing*, Monterrey, Mexico, 1995.

[108] D. Tsichritzis and A. Klug. *The ANSI/X3/SPARC DBMS Framework*. AFIPS Press, 1978.

[109] E.E. Villarreal and Don Batory. Rosetta: A Generator of Data language Compilers. In *1997 Symposium on Software Reuse*, 1997.

[110] Drew Wade. SQL3/OQL Merger. http://www.jcc.com/sql_odmg_convergence, May 1996.

[111] Michael Werner. Pinned Views. ftp://ftp.ccs.neu.edu/pub/people/lieber, March 1995.

[112] Michael Werner. Visitors, Access Paths and Semantics (VAPS). In E. J. Neuhold D. K. Hsiao and R. Sacks-Davis, editors, *OOPSLA'97 Workshop on Object-oriented Behavioral Semantics*, volume TUM-I9737. Technische Universität München, sep 1997.

[113] Rebecca Wirfs-Brock, Brian Wilkerson, and Lauren Wiener. *Designing Object-Oriented Software*. Prentice Hall, 1990.

[114] Carlo Zaniolo, Stefano Ceri, Christos Faloutsos, Richard T. Snodgrass, V. S. Subrahmanian, and Roberto Zicari, editors. *Schema and Database Evolution in Object Database Systems*, pages 411–495. Morgan Kaufmann, San Francisco, CA, 1997.

[115] Roberto Zicari. A Framework for Schema Updates in an Object-Oriented Database System. In *Proc. 7th Intl. Conf. on Data Engineering*, Kobe, Japan, April 1991. IEEE.