The Round Trip Problem: A Solution for the Process Handbook

by

Frank Yeean Chan

Submitted to the Department of Electrical Engineering and Computer Science
in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

at the
Massachusetts Institute of Technology
May 1995

Author ______________________________
Department of Electrical Engineering and Computer Science
May 30, 1995

Certified by [illegible]
Professor Thomas W. Malone
Thesis Supervisor

Accepted by [illegible]
F. R. Morgenthaler
Chairman, Department Committee on Graduate Theses

The Round Trip Problem: A Solution for the Process Handbook

by

Frank Yeean Chan

Submitted to the Department of Electrical Engineering and Computer Science
on May 1995 in Partial Fulfillment of the Requirements for the Degrees of

Bachelor of Science in Computer Science and Engineering
and
Master of Engineering in Electrical Engineering and Computer Science

Abstract

This thesis addresses a problem which arises when different computer systems attempt to share data through an interchange format. Let us suppose that the format has a provision for storing system-specific data. After one system exports a file in this format, a second system importing the file may find it inconvenient or impossible to accommodate data specific to the first. While this loss of information is understandable in a one-way exchange, the real problem occurs when the data is reimported into the originating system. When a file is exported by the second for import by the first, we will have lost any system-specific data with which we began. This is the "round trip problem," the loss of information during the exchange of data between two systems.

In this thesis, I describe the round trip problem as it is encountered in the Process Handbook project, where process data is exchanged between handbook prototypes using the Process Interchange Format (PIF). I propose a solution based on the concept of the companion file. I then describe the PIF toolkit, a code library that serves as the foundation for translators that apply this solution. Finally, I demonstrate the feasibility of the solution through PIF translators built using the toolkit.

Thesis Supervisor: Dr. Thomas W. Malone
Title: Patrick J. McGovern Professor of Information Systems
Director, Center for Coordination Science

I dedicate this thesis to my mother and father, Mei Mei and Chun Fai Chan. They were always there when I needed them, not once turning their backs on me. I will be forever indebted to them for their tremendous sacrifices for my education and happiness. They have forsaken a better car, a better house, and better sleep in the hopes of having a better son. When I had no other motivation, they were my undying inspiration. On my round trip from home to MIT and back, I only care to preserve my family attribute.

Trust me. I know what I'm doing. – Sledge Hammer (1986-1988)

Acknowledgments

I thank Thomas Malone, my thesis supervisor, for guiding and supporting my research. Besides introducing me to the round trip problem, I also credit him and his center with providing me a more complete MIT experience. He has given me not only the freedom to pursue an intellectual challenge but also the opportunity to present a practical solution that others may find useful.

This thesis builds upon the research of many people, but I would particularly like to acknowledge Tom and Jintae Lee for their work on the handbook project, PSV, and PIF. I wish the PIF working group success in establishing an academic and industry standard.

I am grateful to my distinguished panel of thesis reviewers: Tom Malone, Gregory Pal, Erfan Ahmed, Kevin Dye, George Wyner, and Parag Gupta. Besides putting up with incomplete drafts of hieroglyphics, they have been critical in helping me develop my ideas from a porous partial solution. Without their input, especially that of George, this paper would not even resemble "a solution for the Process Handbook."

George Wyner has been helpful to me in so many aspects, from theoretical to personal issues. I began working at CCS with George at about the same time his son Samuel was born, and I doubly thank him for his precious time then. Stephen Buckley's assistance and refreshing sense of humor have made CCS a more enjoyable place to work.

So no one told you life was gonna be this way.
Your job's a joke, you're broke, your love life's D.O.A.

To my *Friends* in the pool, I have enjoyed our camaraderie. Through the "smoke and mirrors," I have learned just as much from what we haven't accomplished as what we have. It has been a distinct pleasure to work with Narasimha Rao and Greg Pal. I also have appreciated Kevin Dye for his companionship in cyberspace. By the power vested in me, I bestow upon him an honorary hackership.

To Erfan Ahmed and Naved Khan, the other two in the three tier architecture, an end to misery is better than misery without end. Once we have earned our degrees, no one can take this accomplishment away from us. I give an Olympic thanks to Erfan, who made numerous modifications and bug fixes to the DB-Layer for the PIF translators. May his next inheritance be something he actually wants – or at least something that works.

I am the Great Cornholio! I need TP for my bunghole! – Beavis

Table of Contents

1 Introduction

Given an interchange format for computer data of a certain domain, its users are likely to run into the following problem. Suppose a user wants to share data between two different computer systems, X and Y, which can read from and write to this format. This may involve the use of a separate translator program that works with each system's native files. After creating data with system X, the user exports this data to the interchange format. By its very nature, an interchange format for a particular industry or field of research is designed to contain generalized data applicable to many, if not most, systems. So, on the export from X, data specific to system X may be left out. Let us optimistically suppose that the interchange format is able to accommodate such data. Upon importing this first interchange file into system Y, however, only that which is relevant to Y will be translated. Thus, some of the system X data will be ignored.

Though not ideal, this is to be expected. While the phrase "lost in translation" summarizes the difficulties of any one-way exchange, the real problem occurs on the data's return trip from system Y to system X. After using system Y, the user exports this data to a second interchange file. This will contain general modifications made in system Y and, perhaps, some Y-specific data. Then, when this file is translated back into system X, the user will have lost any X-specific data with which he or she began. It is this loss of information which we characterize as the "round trip problem." In any sharing of data through an interchange format that completes a cycle between multiple systems, we would like to be able to preserve as much system-specific data as possible.

This paper describes a solution to the round trip problem for the Process Handbook project at MIT's Center for Coordination Science (CCS). The goal of this project is to help organizations redesign their existing processes and to invent new organizational processes that take advantage of information technology. The round trip problem arises when using the Process Interchange Format (PIF) to exchange data between different handbook prototypes. Originally developed for the handbook, PIF is a translation language for sharing business process descriptions among diverse representations.

In the next section, we provide background material on the handbook, PIF, and the relationship between the two. In Section 3, we analyze the round trip problem in PIF and categorize exactly what information can be lost in a PIF translation. We can then present our solution framework, as well as identify the solution's requirements and assumptions.

Section 4 describes the PIF toolkit, the foundation for translators that preserve system-specific data in exchanges. We also outline the toolkit's import and export algorithms, the steps taken to solve this problem. In Section 5, we discuss the implementation of translators based on the toolkit for various systems. Prototypes of the handbook are the primary testing grounds for the solution. Finally, we arrive at some conclusions concerning our solution framework and see how it could be applied to other situations.

2 Background

2.1 The Process Handbook

2.1.1 Project Overview

This project seeks to collect examples of business processes and represent them in an electronic handbook (Malone et al 1993). This handbook will help in managing organizations by displaying alternatives to how a given process could be performed with their relative advantages. This on-line tool allows the user to compare and contrast related processes based on factors such as cost, time, etc. By learning how different organizations perform similar processes, one can redesign existing processes or invent new processes. With our methodology for representing organizational practices and our extensive collection of examples from the field, our goal is to provide a firmer theoretical and empirical foundation for tasks such as enterprise modeling and business re-engineering.

The project's key intellectual challenge is developing techniques for representing these processes. Our research on this subject draws upon ideas from computer science about inheritance and from coordination theory about managing dependencies. Our goal was to develop a handbook where a user would not have to analyze every new organizational process "from scratch." For example, assuming the handbook had a representation of a generic "sales process," someone studying a specific sales process should be able to focus on what is novel about the new situation. Thus, we have developed a language that explicitly represents the similarities and differences among related processes. This facilitates the display of alternatives which might be appropriate for a given scenario.

The novelty of the handbook's approach lies in its representation of processes at varying levels of abstraction. The handbook is arranged in a hierarchy of increasingly specialized processes, with the more generic at the top of the tree[1] and the more specific toward the bottom. Similar to object or type hierarchies in object-oriented programming and knowledge representation, processes at any level "inherit" attributes or characteristics from higher levels by default. In addition, a process can add or change their own

[1]More precisely, this "tree" is a directed acyclic graph (or "dag") since an activity can be a specialization of multiple activities. In other words, a node in the hierarchy can have multiple parents.

attributes, perhaps for more specialized processes to inherit. The advantage of this approach is that it empowers users to only represent what is different about a new situation. Also, the amount of work necessary to document a new process is significantly reduced from having to start at "ground zero." Most importantly, this specialization hierarchy gives the handbook a structure where numerous examples collected from the field can be organized effectively.

Additionally, the handbook represents the concept of process decomposition in a second, orthogonal hierarchy. In contrast to the "is a kind of" relationship linking nodes in the specialization hierarchy, parent and child nodes have a "is a part of" relationship in this decomposition hierarchy. To see the complementary relationship between these two hierarchy trees, we look at a small example concerning sales activities in Figure 1 (a simplified version of Figure 1 from Malone et al 1993). At the first level, the more general "Sell product" activity specializes into the "Direct mail sales" and "Retail storefront sales" activities. Each of these three has, in turn, a set of subactivities representing its decomposition.

Though reminiscent of object-oriented inheritance in computer science, inheritance in the handbook is uncommon in the following two ways. First, upon the creation of a specialization, say "Direct mail sales," the new activity inherits not only the simple textual attributes but also the complex decomposition attribute of the parent.[2] As expected, changes can be made to a child's decomposition, such as the replacement of "Identify prospects" with "Obtain mailing lists" in "Direct mail sales." Second, a user can represent the omission of a subactivity in a specialization, such as the lack of need to identify prospects in "Retail storefront sales." In a more rigid, traditional object hierarchy, this is not possible. Every attribute of a base object is relevant in more specialized objects. For example, every attribute of a "Vehicle" object, such as its weight, should pertain to a specialized object like "Automobile." A current topic of research is a more formal description of handbook inheritance and its differences, if any, from traditional notions of inheritance (Wyner and Lee 1995).

[2]With multiple inheritance, the more accurate term should be "parents." The composition of multiple decompositions (or, in more general terms, the semantics of multiple inheritance) is a topic that has not been investigated fully. Currently, we use a simple "union" operation, supplemented by human interaction to add or delete process steps and reconnect dependencies (described later).

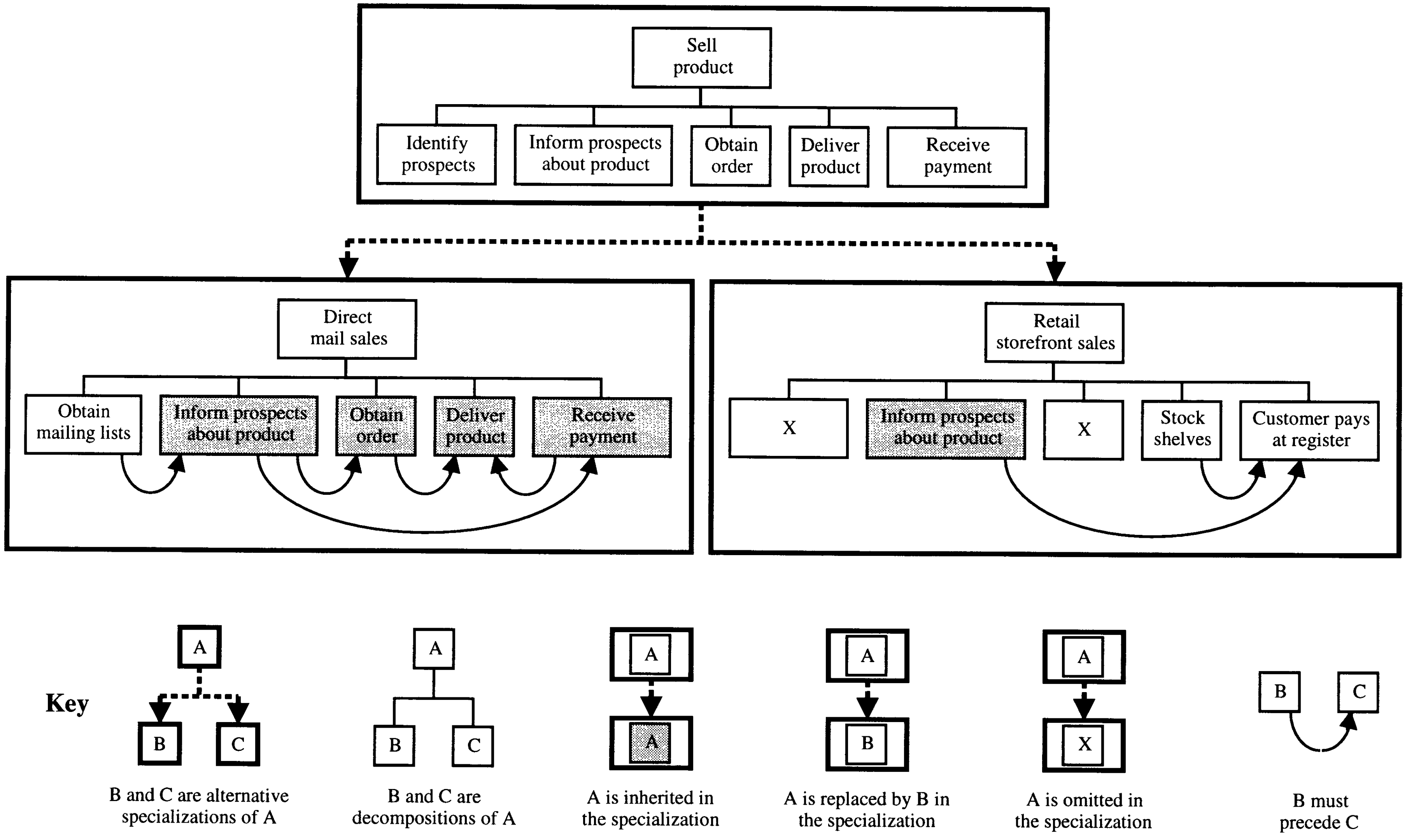
Sell product
Identify prospects
Inform prospects about product
Obtain order
Deliver product
Receive payment
Direct mail sales
Obtain mailing lists
Inform prospects about product
Obtain order
Deliver product
Receive payment
Retail storefront sales
X
Inform prospects about product
X
Stock shelves
Customer pays at register
Key
B and C are alternative specializations of A
B and C are decompositions of A
A is inherited in the specialization
A is replaced by B in the specialization
A is omitted in the specialization
B must precede C

Figure 1: The relationship between the specialization and decomposition hierarchies in the handbook.

Related to inheritance is the concept that changes made to the handbook must have a context. As the handbook classifies processes at various levels of abstraction, the context represents the level of abstraction at which a change should be made. This context is important because it determines the scope of activities that are affected when the change is propagated by inheritance. If we would like to further decompose the subactivity "Deliver product," we must first determine whether this decomposition is specific to "Direct mail sales," general to all sales processes, or even, perhaps, global to all activities where a product is delivered. For example, the subactivity "Track shipment" only makes sense in the context of product delivery for mail order sales, but "Confirm delivery" might be more appropriate at the generic "Sell product" level.

Though not covered in depth here, much of the research on the handbook involves coordination theory or the management of dependencies between activities. In Figure 1, the two specialized sales processes have prerequisite dependencies, a constraint that one subactivity must take place before another. Adding a dependency is an example of a change to the handbook that requires a context. For example, in the context of "Direct mail sales," the payment must be received before the product is shipped. In other contexts, such as a promotional offer where the customer has a free trial period, the product may be shipped before payment is received. Thus, these dependencies are not displayed at the generic "Sell product" level because the constraints on delivering and paying are not universal to all sales processes. They are also not an intrinsic property of delivering a product, so we consider the dependency to be a property of, or belong to, the ancestor activity instead of the participating activities.

With the goal of allowing the user to compare alternatives, related specializations in the handbook can be grouped together to form a "bundle." Then, given this refinement, comparisons will be most appropriate between activities in the same bundle. For a given activity, a bundle can be thought of as an axis or direction along which further specializations can be added. Displayed graphically in a tree, these bundles appear as an intermediate layer between an activity and its specializations. In Figure 2, "Direct mail sales" and "Retail storefront sales" belong to the "How sold?" bundle. "Sell product" also has bundles grouping specializations based on what is sold or to whom the product is sold. As an example with multiple parents, "Sell clothes by mail order" is of special interest. It should inherit the attributes and subactivities of both the "Sell clothing" and "Direct mail sales" activities.

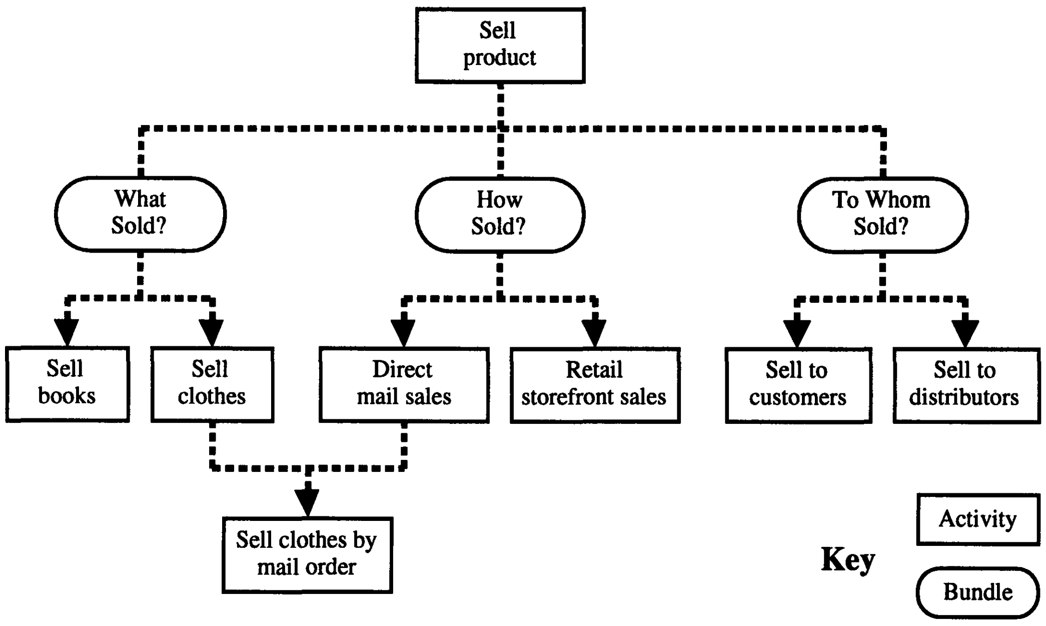

Figure 2: The bundles of specializations for "Sell product."

2.1.2 Architecture Requirements[3]

Before drawing up any plans for a system architecture, we should analyze the variety of usage scenarios we envision for the handbook. Initially, a group of qualified individuals (the "founding fathers" of the handbook) will organize their collective knowledge about basic business and coordination processes into the aforementioned specialization and decomposition hierarchies. Given this skeleton tree of generic activities such as "Sell clothes by mail order," a field investigator studying a specific situation (e.g., Lands End) will first add his or her description as a new leaf (specialization) where appropriate. Then, he or she will edit the inherited decomposition of the new process to reflect its uniqueness. At some later stage, a qualified editor will review and approve new additions to the handbook. Finally, with a populated handbook, one should be able to learn from and compare actual process examples, perhaps even through a workflow viewer or a simulation engine.

The handbook can be viewed as a collection of services, and our architecture must accommodate the needs of a diverse set of users. Still, all these different needs impose a few common requirements. First, our system needs a storage facility that can handle our

[3]Narasimha Rao, a former research assistant at CCS, performed much of the investigation of the architecture requirements.

process representations. This database package must have a programming interface whereby "process inheritance" could be implemented. Though an object-oriented database may seem like a natural choice, inheritance in the handbook has some additional caveats as discussed earlier. Furthermore, the arbitrary specialization of activities requires a dynamic object system, either built in to the database or simulated by meta-objects. Lastly, a thorough process description consists of much more than just its decomposition, so we should be able to include types of data other than just plain text (formatted text, diagrams, pictures, tables, links to longer reports, etc.).

Second, we would like the development of the handbook to operate in a distributed environment. Multiple users will access the database at the same time, and they will make changes interactively as well as remotely.[4] Having a central repository facilitates the maintenance of the data and the review of handbook submissions. Those in the field mapping new processes will most likely be working on stand-alone (non-networked) environments, such as a personal computer laptop. However, they will still need to connect to the central server from time to time. After having previously downloaded some subset of the handbook data, field investigators will eventually replicate their new entries back to this repository. This raises the standard issues of data consistency and version management, mainly when two users separately modify the same data and then hope to combine their changes.

Third, all users of the handbook must have tools to view and edit process descriptions. These tools should be able to operate on any size subset of the database. They should also have some provision for displaying and storing the heterogeneous data found in complete process descriptions. Editors must be able to display changes from and write changes to the repository. A seemingly simple edit of a high-level process may affect many other activities due to inheritance calculations and propagations.

Most importantly, these viewers and editors must allow navigation within and between the two types of process hierarchies. There are two different patterns of handbook usage that lead to this requirement. Those browsing the handbook will likely begin at the root of the specialization hierarchy (the most generic activity) to find an activity of interest, then investigate its decomposition (a top-down approach). On the other hand, those

[4]Remote collaboration is especially important in the handbook project since the four principal investigators (Profs. Malone, Crowston, Lee, and Pentland) are conducting research at different universities across the United States of America (M.I.T., University of Michigan, University of Hawaii, and University of California at Los Angeles, respectively).

entering in a new process may prefer to first think in terms of its steps or subactivities, then find a place in the specialization hierarchy where it belongs (a bottom-up approach).

Finally, as a practical consideration, our system should be designed to be usable on laptops given the usage scenario for field investigators. First, this implies that, with the current state of computer technology (i.e., with only personal computer laptops), a handbook implementation should not solely exist on workstation platforms (e.g., UNIX). Second, as mobile computing is currently no complete substitute for desktop computing, we must be sensitive to the issues of screen real estate and processing power, especially with inheritance computations.

2.1.3 Prototype Implementations

Initially, we sought a single system on which to build the handbook. The first major handbook prototype was built by Chris Dellarocas, a CCS doctoral student, in Kappa-PC, a knowledge-based software tool from Intellicorp. As a "proof of concept," this prototype demonstrated much of the intended functionality in process inheritance and hierarchy browsing. However, its capabilities for handling large numbers (hundreds) of processes and structured text descriptions were limited. Also, the Kappa-PC tool had no provision for multi-user access other than passing around the prototype's database from user to user like a baton.

George Wyner, also a doctoral student at CCS, then built the second major prototype in Lotus Notes, a groupware software package, to answer some of these shortcomings. With a framework designed for collaboration, this version easily allowed contributions from all handbook researchers. Remote participants replicated their work via modem or network connection back to the Notes server physically residing at our research center. Plus, each entity (activity, bundle, or dependency) was treated as a document generated from a form or template. Because each document could then hold "rich" (formatted) text and embedded graphics, they more strongly resembled full process descriptions. Though a user could browse through outline views of the specialization and decomposition hierarchies, this prototype did not have inheritance, unfortunately. In other words, activities did not inherit the properties and subactivities of their ancestors.

With time, the handbook architects concluded that no single, existing system could meet all our architecture requirements. Nevertheless, for the handbook to be most useful, it would need capabilities in a broad spectrum of areas: object-oriented inheritance, distributed database access, semistructured multimedia editing, hypertext, flow-charting, and others. Without the human resources of a commercial software house, we could not afford to build such a handbook "from the ground up." We then pursued a "componentware" approach in the hopes of making the most of existing technology, as opposed to trying to compete with it.

We chose to develop a three tier architecture (Figure 3) with the following components: a process repository, various handbook applications and graphical user interfaces, and a database access layer for application developers. Erfan Ahmed, a CCS research assistant, has thoroughly analyzed this decision and described an implementation of the intermediary layer (Ahmed 1995). Having this middle layer (also known in some circles as the business logic) gives us two advantages. First, for any component, we can choose to build upon the packages that have the desired functionality in a specific area. The two previous prototypes, each built upon a single software development tool, were constrained by having the database inseparable from the application. With the choice of one no longer binding the choice of the other, we can hope to have the best of both worlds. Second, by incorporating the logic of process inheritance within the "middleware," we can eliminate code duplication and ensure consistency in inheritance operations across all editors.

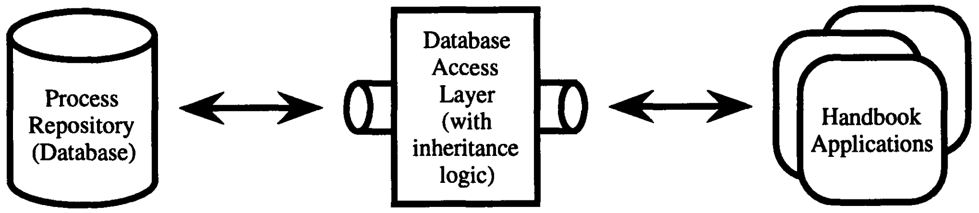
Process Repository (Database)
Database Access Layer (with inheritance logic)
Handbook Applications

Figure 3: The three tier handbook architecture (Ahmed 1995).

On the down side, the savings in development work from using existing products is lessened by the additional effort needed to integrate these multiple components. Also, a prototype created from this architecture will most likely not have the tightly integrated look of the original prototypes created from "all-in-one" systems.

2.2 The Process Interchange Format

2.2.1 Motivation and History

Thus, instead of the daunting task of building a single, monolithic system to support all possible uses, our research team has chosen to extend existing systems with handbook functionality. A key element of such a plan is a mechanism for exchanging process descriptions between different tools. Thus, we began developing the Process Interchange Format (Lee and Yost 1994), an effort led by Jintae Lee, one of the principal investigators in the handbook project. This approach allows us to view the handbook as a collection of specialized tools bonded by the glue of PIF. Just as importantly, we aimed to preserve the data entered in previous prototypes for our current and future tools.

As we proceeded, we realized that a standard format of process representations could be valuable to others. The business process descriptions found in our project have a wide variety of applications: process re-engineering, workflow management, simulation, etc. With different kinds of systems exchanging data through the PIF, these systems could interact much more easily. Consequently, we formed a working group in October 1993 consisting of representatives from several universities and companies to develop the format (the most active being University of Hawaii, Stanford University, University of Toronto, and Digital Equipment Corporation).

There are two goals for this PIF project. First, we sought to automatically exchange process descriptions between systems of the working group members. To achieve this goal, each system only needs to have translators to convert data between its native format and the common interchange format. For "n" number of systems, the number of translators (or translation directions) will only grow linearly (n*2: for each $system_i$, PIF-->$system_i$ and $system_i$-->PIF) instead of quadratically (n*(n-1): for each $system_i$, $system_i$-->$system_j$, where $i \neq j$). The second and larger goal is to arrive at a foundation for what might eventually become a standard for exchange among researchers, corporations, and users of commercial products. By supporting interchange between different process representations, we also promote the sharing of related work, and this would be beneficial to all parties.

To develop PIF, the group members began by examining their respective projects to find commonalities for an interchange format. We analyzed the types of objects represented in process descriptions, such as activities and resources, along with their attributes and relationships. In addition to our handbook, the two other most prominent systems were the Spark project at Digital Equipment Corporation and the Virtual Design Team (VDT) project at Stanford University. The Spark project, represented by Gregg Yost, aims to create a tool for creating, browsing, and searching libraries of business process models. The VDT project, represented by Yan Jin, aims to model, simulate, and evaluate process and organization alternatives.

After the group explored the similarities in their systems, the main challenge was to define the minimal ontology (specification of our object model) of constructs in existing business process descriptions which is still rich enough for meaningful exchange between various systems. Some, like the notion of an "activity," were present in all three systems, so these were good candidates for the common ontology. On the other hand, most of those that existed in only one system, such as the handbook's "bundle," were considered too project-specific. More interestingly, groups sometimes had different names for similar (if not identical) objects; "actor" and "agent" can both refer to a human resource. Similarly, we occasionally had different interpretations for an object of a particular name.

These more tangible objects did not present as much of a problem as relationships between objects. The group had to determine if any given relationship should be an attribute of one of the entities involved (i.e., a link or "pointer" from one object to another) or a separate entity in itself. For example, if an activity "requires" a certain resource, this may be represented as a property of the activity, a property of the resource, or a "relation" object. These differences in representation are known as "category differences" (Tse 1991). For the interchange format, we decided to package a relationship as a separate object only if we could envision it having attributes of its own. Even with this decision for PIF, the relationship may be represented differently in any particular system. This means that translators may have to convert between these different approaches when mapping a file of PIF objects to a system (or vice versa).

2.2.2 Description

The working group finalized the first version (1.0) of a PIF ontology in November 1994 (Lee and Yost 1994). The philosophy of the group was to start with the minimal ontology and only expand through proposals accompanied by concrete examples illustrating the need. (The ontology is undergoing further refinement and will likely be extended for a second version in 1995.) This "core" ontology is arranged in a class (or type) hierarchy (Figure 4). At the root is the ENTITY class, which defines some basic attribute slots such as "Name" that are relevant to all PIF entities. Each class has a set of attributes defined specifically for itself, and, as in traditional type hierarchies, it inherits the attributes of its parent. For example, we consider a decision to be a subclass[5] (or subtype) of activity, so a DECISION type inherits all the ACTIVITY attributes. An actual PIF process description consists of a file of these kinds of objects.

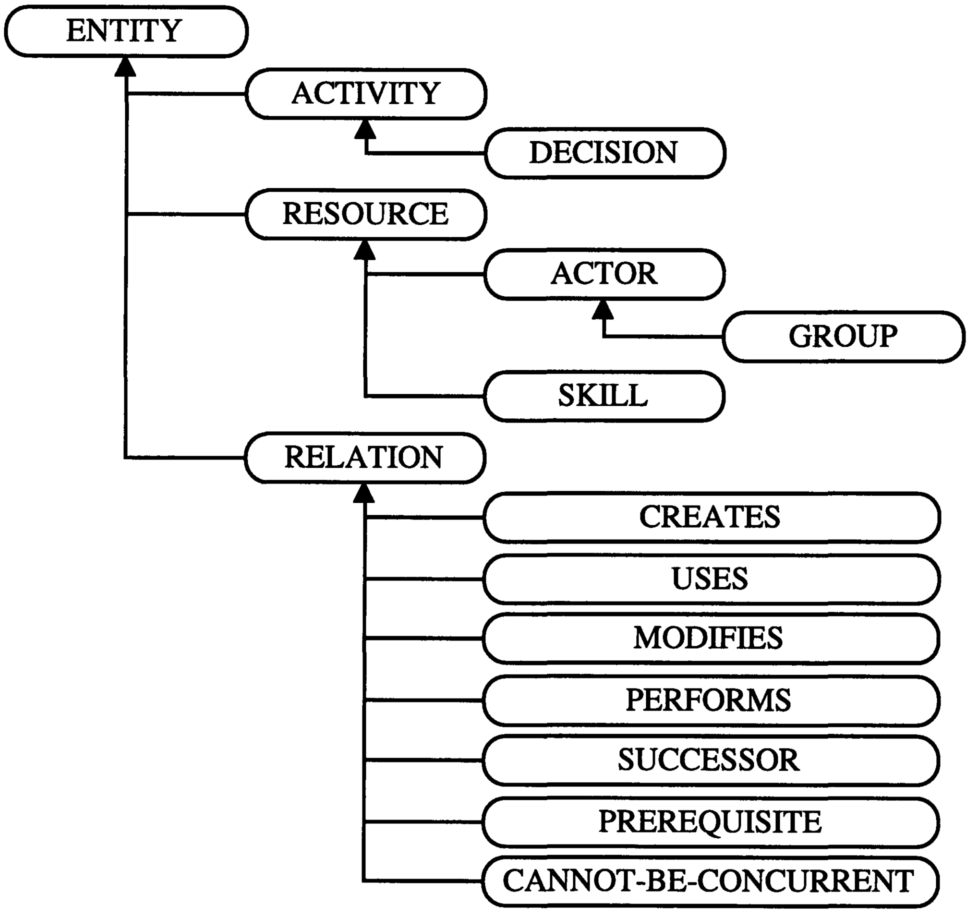

Figure 4: The PIF class hierarchy (Lee and Yost 1994).

In addition to the core set of object types, PIF provides a framework for extending this set of types to include additional information needed in specific applications. No interchange format, including PIF, is likely to completely meet the needs of the systems that use it. On one hand, the supplemental data may be system-specific attributes for a core class, such as graphical layout information for how an activity should be drawn. On the

[5]We refrain from using the term "specialization" here, reserving it for the Process Handbook. Of course, we still consider a decision to be a special kind of activity. Whether or not to treat process specialization and class subtyping as the same is discussed in section 5.2.2. For now, we maintain the distinction.

other hand, this data may also come in the form of extensions to the ontology (i.e., new subclasses). For example, in light of the handbook, we would like to represent objects of the bundle type and various dependency types in a PIF file.

To accomplish this, PIF employs the Partially Shared Views (PSV) scheme for communicating among groups that use different type hierarchies (Lee and Malone 1990). For our framework, PSV attempts to maximize the level of information sharing among groups that have extended the core ontology. With many systems requiring extensions to various parts of the core class hierarchy, differences in granularity will be inevitable. Some will find it necessary to represent multiple types of actors, while others will find the core ACTOR class to be sufficient for their needs.

Though the details of PSV are explained elsewhere, here is the basic idea: an object of an extended type that is not recognized (not "in view," so to speak) should be translated as an object of the nearest parent type that is in view.[6] Unless two systems happen to share a view of a set of subtype extensions, this parent will be a core PIF type. Suppose that a system which has a type extension of ACTOR named EMPLOYEE exports a process description. Upon importing this PIF file into a system that does not support this extension, EMPLOYEE objects should be converted to ACTOR objects. Thus, this mechanism allows objects to be translated with as little loss of meaning as possible, as an employee is just a kind of actor. It also brings an "expand on need" philosophy to PIF, preventing the core ontology from being infested with types that are not used in practice but only seem potentially beneficial.

The syntax of PIF is built upon the Knowledge Interchange Format (KIF) (Genesereth and Fikes 1992), a language designed for sharing knowledge among computer systems. Much effort has gone into making KIF a good general-purpose format, and we feel it is as good as, if not better than, anything else. For our purposes, any syntax that can specify classes and instances (object types and actual objects) as well as their attribute types and values will suffice, and KIF is well suited to do that with little overhead.

In particular, PIF adopts Ontolingua's frame syntax, an extension of KIF for representing object-based knowledge. Ontolingua itself is a system for analyzing and

[6]PIF version 1.0 does not support multiple inheritance. However, it is possible that a future version of PIF will. With multiple inheritance, a more complicated PSV mechanism will be necessary to translate objects into their nearest parent types in view.

translating ontologies through higher-level KIF constructs (Gruber 1993). Along with the grammar for PIF syntax, version 1.0 of the core ontology in frame syntax can be found in the appendix. As an example, the class frame for ACTIVITY follows (Frame 1). "Own-slots" are attribute values of the frame itself, but "template-slots" define attribute slots for all instances of the frame. The specific meanings of these template-slots is discussed later in section 4.2.2, but their general purpose is to specify constraints on actual ACTIVITY objects. Thus, both classes and instances can have own-slots, but template-slots are only allowed in classes. Though not displayed in this example, any actual values in a template-slot are inherited. Suppose we wanted all ACTIVITY instances to have a default Status value. This value would be part of the Status template-slot in the ACTIVITY class.

```
; Frame 1: An example class frame, the ACTIVITY class.

(define-frame ACTIVITY
      :own-slots (
            (instance-of       CLASS)
            (subclass-of       ENTITY)
      )
      :template-slots (
            (Component         (slot-value-type  ACTIVITY))
            (Precondition      (slot-value-type  RESTRICTED-KIF-SENTENCE)
                               (maximum-slot-cardinality     1))
            (Postcondition     (slot-value-type  RESTRICTED-KIF-SENTENCE)
                               (maximum-slot-cardinality     1))
            (Status            (slot-value-type  SYMBOL))
      )
)
```

Though KIF is based upon Lisp (Steele 1990), this should not be a cause for concern to those not accustomed to the language. We are mainly using Lisp to provide a structure for our information. In addition to the number, symbol, and string literal types, PIF allows sentences of a subset of KIF, named a RESTRICTED-KIF-SENTENCE in our syntax. The idea here is to provide some of KIF's expressive power without requiring every PIF translator to be able to parse everything in Common Lisp.

2.2.3 Additional Issues

This section is a discussion of other PIF issues that are relevant to either the handbook or the round trip problem.

After much debate, the working group decided that PIF files should only contain instances. In other words, they should not have definitions of any new class types. As with the creation of the RESTRICTED-KIF-SENTENCE, our main motivation was to reduce the complexity of the translators. Allowing new classes in process descriptions would require translators to have a dynamic object system to process the class and attribute type (template-slot) definitions. The group members felt that some potential PIF users, many of whom are from industry instead of academic research environments, would be reluctant to participate if such a system (perhaps Lisp-based) was necessary.

This decision had two immediate consequences for our format. First, translators would still require some class information about ontology extensions. More specifically, in PIF's version of the PSV scheme, they still need a class hierarchy to find nearest visible parents. For example, in order to correctly import an EMPLOYEE object into a system that did not support this type extension, we must at least know that the type was a subclass of ACTOR. Thus, at the beginning of every file of objects, before all the "define-frame" instance definitions, PIF requires a simple "define-hierarchy" construct. Essentially a textual representation of the tree of Figure 4, a sample hierarchy with the EMPLOYEE extension follows in Frame 2.

```
; Frame 2: An example define-hierarchy construct with the EMPLOYEE
;          extension.

(define-hierarchy
      (ENTITY
            (ACTIVITY
                  DECISION
            )
            (RESOURCE
                  (ACTOR
                        GROUP
                        EMPLOYEE
                  )
                  SKILL
            )
            (RELATION
                  CREATES
                  USES
                  MODIFIES
                  PERFORMS
                  SUCCESSOR
                  PREREQUISITE
                  CANNOT-BE-CONCURRENT
            )
      )
)
```

The difference between full class definitions and this single hierarchy definition is the amount of information known about type extensions. For the PSV scheme, it is not necessary to know what kinds of slots EMPLOYEE objects have. Instead of parsing and storing a full EMPLOYEE frame like the ACTIVITY example (Frame 1), all a translator needs to know is that an EMPLOYEE is a subtype of ACTOR. This uncomplicated, minimalist approach meets the need of finding an ancestor. It also suits the working group's desire to have PIF files self-contained, so they can be shared without additional information. Thus, PIF allows type extensions but not full type definitions. However, we will later reintroduce these definitions in section 4.2.2 for the solution to the round trip problem.

The second consequence of not allowing classes is that attribute values must be "flattened." That is, an object must explicitly contain all its attributes and values, including those inherited without modification from its class and ancestor classes.[7] Originally, a possible alternative would be to have "non-flattened" attributes, where an object only contains those attributes and values that are different from its parent classes. In some systems, it may be useful to assign a default value for a particular attribute in a class, such as a default "Salary" value for the EMPLOYEE type. Then, actual instances of the EMPLOYEE type and subtypes would not have to explicitly contain this attribute unless they had a different "Salary" value.

However, without class definitions, a translator has no way of determining what values should be inherited for any given instance in a PIF file. For example, when the EMPLOYEE object is imported into another system, there is no way of knowing the first system's default value for "Salary." Therefore, an EMPLOYEE object should have all such inherited values when it is exported to PIF since the class definition will no longer be available. Having flattened attributes also eliminates the need for PIF translators to perform inheritance operations, partially a result of not having or requiring a dynamic object system.

Of final note, our framework has a provision for storing any attributes of an object that are not part of its type definition. Every PIF entity can have an additional "User-Attribute" which itself consists of attribute name and value pairs. As opposed to type extensions, this is an alternative method of including system-specific information. In our experience, we have used this method either before the proposal of a subtype or when we do not feel that the object warrants such a distinction. For example, this ordinary ACTOR object has two extra fields (Frame 3).

```
; Frame 3: An example instance frame, an ACTOR object with two user
;          attributes.
```

[7]Currently, no class in PIF 1.0 has a default value. If a system's classes do not have any such values either, this issue of "flattening" does not exist since instances will have no values to inherit. This implies that all instances will be necessarily flat.

```
(define-frame 720909171700FYC
      :own-slots (
            (instance-of      ACTOR)
            (Name             |Frank Chan|)
            (User-Attribute   '(Age             22)
                              '(E-Mail-Address  "mickey@mit.edu"))
      )
)
```

Besides the storage of "ad hoc" attributes, another use of this slot involves our PSV scheme. Suppose that a system's EMPLOYEE class defined a couple of slots, say "Medical-Plan" and "Salary." An EMPLOYEE object in this system appears in Frame 4. Upon translating to a different system that did not share the view of the EMPLOYEE type, the "Medical-Plan" and "Salary" attributes would be relocated to the user attributes repository as displayed below. The converted ACTOR object in the second system is displayed in Frame 5. Even if an importing system cannot make use of the these attributes, we aim to preserve them in the hopes that the translator can add them back to the process description when exporting back into PIF. Unfortunately, not every system has a place for these arbitrary attributes, and this is part of what leads us to seek a more universally applicable solution to the round trip problem.

```
; Frame 4: Before PSV conversion, an EMPLOYEE object in a system with
;          the EMPLOYEE type extension.

(define-frame 720909171700FYC
      :own-slots (
            (instance-of      EMPLOYEE)
            (Name             |Frank Chan|)
            (Medical-Plan     "Prudential")
            (Salary           1.99)
      )
)

; Frame 5: After PSV conversion, the same object as an ACTOR in a system
;          without the EMPLOYEE type extension.

(define-frame 720909171700FYC
      :own-slots (
            (instance-of      ACTOR)
            (Name             |Frank Chan|)
            (User-Attribute   '(Medical-Plan    "Prudential")
                              '(Salary          1.99))
      )
)
```

3 The Round Trip Problem in PIF

As suggested earlier, part of the motivation for having the "User-Attribute" repository was to preserve all attributes of the original objects not in view. Just as the PSV scheme aims to minimize the loss of meaning in a single translation between representations, one of the objectives of the PIF project is to retain as much information as possible over multiple translations. We aim to share process descriptions such that:

- Upon the import of a process description, we would like to represent any untranslatable data in a viewable manner. Even if the data only appeared as a textual comment, this will allow a user to see the problem and, perhaps, complete the translation manually if possible and desired. Our intent is to have a graceful degradation in the quality of translation.
- For the following export back to a PIF file, we would like to save these uninterpretable parts in the system so that the translator can later add them back to the description, restoring those attributes not explicitly edited.

After a description of the problem and solution framework, we will discuss quite a few issues including preserving the identity of objects, naming conflicts, multiple imports and exports, and the practicality of the solution. We first focus on a general strategy for solving the heart of the problem and deal with these secondary issues later.

3.1 Two Aspects of the Problem

3.1.1 Loss of Attributes

Given the context of the Process Handbook, we can now look at a particular scenario with two systems, a graphical viewer and a workflow management tool (Figure 5). Suppose that in the graphical viewer (abbreviated as V), a handbook user models his sales organization by creating the three familiar activities: "Sell product" and two specializations, "Direct mail sales" and "Retail storefront sales." This graphical viewer allows him to neatly position these activities in a tree-like structure. Now, the user wishes to use the workflow management tool (abbreviated as W). On export to PIF, the graphical layout coordinates of each activity (system-specific data) are included as "user attributes."

While importing these objects into system W, the system-V layout information is ignored, as the PIF translator would only be looking for system-W attributes. If we are fortunate, there will be some way to store foreign user attributes in system W. However, we will proceed without making this assumption. In the workflow tool, the user creates an actor, "Frank," and experiments with assigning "Frank" to different sales tasks. After determining that "Frank" is best at "Retail storefront sales," this user decides to return to the viewer to create some further specializations of retail sales processes. In loading the second PIF file (exported from system W) into the viewer, the translator finds no graphical coordinates but only some workflow data irrelevant to system V, possibly sales performance statistics of "Frank." Hence, in the round trip from V to W back to V, the user has lost his layout information.[8] One can also see that if the user were to return to system W through a third PIF file, the previous system-W data would also be lost.

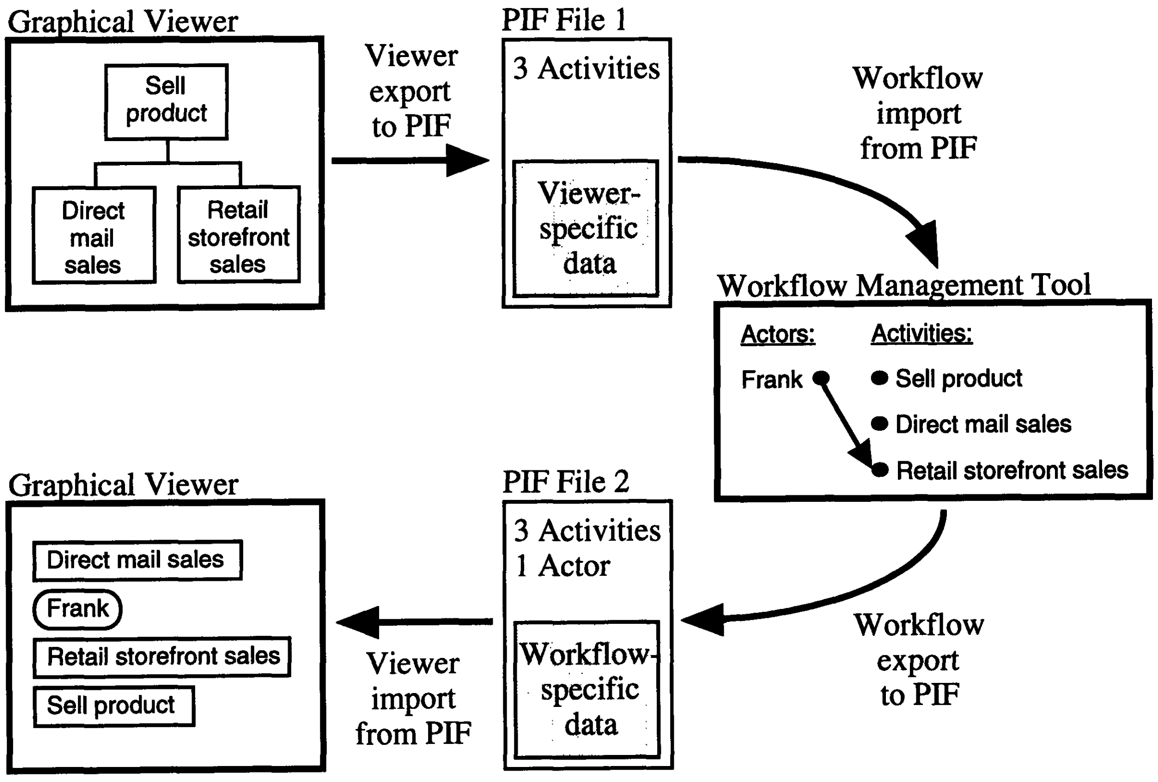

Figure 5: A scenario of the round trip problem in the Process Handbook.

At first glance, one might propose that all systems be able to store arbitrary attributes for all types of objects to solve this problem. Besides the disadvantages of that

[8]For this exercise, we will ignore the possibility that the graphical viewer has an intelligent algorithm for laying out objects based on the handbook specialization hierarchy. The point is that viewer-specific attributes will no longer be present.

approach (to be discussed later in section 3.2), it does not solve a similar yet more severe problem when PIF users attempt to share data after extending the core ontology.

3.1.2 Loss of Type

Revisiting the EMPLOYEE situation from section 2.2.3, suppose we have two systems: X, which supports the subtype, and Y, which does not. While the X-->PIF1-->Y translations take place as expected with PSV conversions to the parent ACTOR type, it is the return trip that yields a disagreeable result. The second PIF file, resulting from a system-Y export, will contain ACTORs (like Frame 5) instead of EMPLOYEEs. When importing the file back into system X, the user will find data that no longer makes use of the type extension. This implies that even if system Y was able to save the EMPLOYEE-specific attributes, these fields would now be considered user attributes of ACTOR objects, strictly speaking. While this does not preclude system X from making use of the attributes, it would be incorrect to restore the EMPLOYEE type as in Frame 4. Without any further information, we cannot assume that just because an ACTOR has "Medical-Plan" and "Salary" data, it was originally an EMPLOYEE. The ACTOR could have these attributes by coincidence.

Thus, the round trip problem in PIF boils down to preserving the type of an object and all of its attributes. Actually, we can look at the type as just another "attribute" of the object except that this particular attribute affects the interpretation of all others. As the core ontology only has the most basic types, we do expect most PIF users to have local type extensions to better suit their needs. At the very least, they will have additional attributes relevant to their respective systems for core classes. Hence, translators will encounter process descriptions which are only partially interpretable. If PIF users only wish to learn from other people's data (i.e., a one-way exchange), the status quo is acceptable. In order for anyone to fully and most easily benefit from running their process descriptions on any other system, however, we ought to find a solution to this problem. Otherwise, a PIF user may have to manually combine results generated from multiple systems.

3.2 The Solution Framework

In developing a solution to prevent the loss of attributes and the loss of type, we recognize that there are other types of knowledge worth preserving in a round trip. For example, there may be various relationships or invariants between attributes of the same or different objects. As a very simple example, suppose one system exported a "Name" and a system-specific "Initials" attribute for an ACTOR instance. If only the object's name was changed in another system, the two attributes may be inconsistent upon return to the original system. As discovered in writing actual translators (section 5.1), redundancy in PIF files should be avoided even if it requires more processing in translation. Here, the first system would be better off not exporting the "Initials" attribute but calculating it upon import of the "Name." In any case, maintaining such invariants is not in the scope of this work. We figure that if this solution provides a way to return data that would otherwise be lost, system architects will be more than willing to check the information's validity.

Before presenting a scheme, we review the main stipulation for any solution. With the goal of universality, the solution should not require any system which imports or exports PIF files to store foreign user attributes, i.e., data specific to other systems. This is too much to expect of those who wish to share process descriptions, especially since it will likely necessitate modification of existing systems. On the other hand, if a system can store this data, PIF translators can make use of this capability.

Even so, there are several reasons why this may be undesirable. First, this may bloat files or databases with data that has nothing to do with the particular system. Though small strings like "mickey@mit.edu" may not pose a problem, encoded picture attributes and other potentially large forms of binary data will. Second, unless it is the intention, there are dangers in allowing one system the right to read and modify data specific to another. While the concern is obviously not security in this spirit of sharing process descriptions, we could no longer guarantee that the interpreted data is not corrupt when written back out. One system may modify an object's attributes in a manner that violates an invariant of another system. If desired, the following scheme can ensure that system Y never touches data specific to system X. Nevertheless, we can still include the foreign user attributes as a comment but draw upon a more reliable source for the return trip.

3.2.1 The Companion File

Our solution to the round trip problem is based on the concept of the companion file. When unrecognized attributes cannot be imported in a system, these companion files serve as repositories for such data in the translation process. Let us now reexamine the round trip scenario of interchange between a viewer (V) and workflow (W) tool in the handbook (Figure 6).

If a tool, the viewer in this case, creates a PIF file for the first time, the translation occurs as before. There is no companion file for this export. Upon import into the workflow tool, the "new and improved" translator prompts the user to leave a companion file of all data which could not be incorporated in system W. In this case, such data would be the layout information used by the viewer (V-data). Then, on the workflow tool's export to PIF, the translator will prompt the user for a companion file which should supplement this system-W data. The data in the companion file will then be merged with the data to be exported, resulting in a complete PIF file with system-specific data for both systems V and W. On the fourth translation of the round trip, the viewer reads the V-data of the PIF file, so the loss of information has been avoided. For any future exports from the viewer, the importing translator prompts the user to leave another companion file, this time containing W-data.

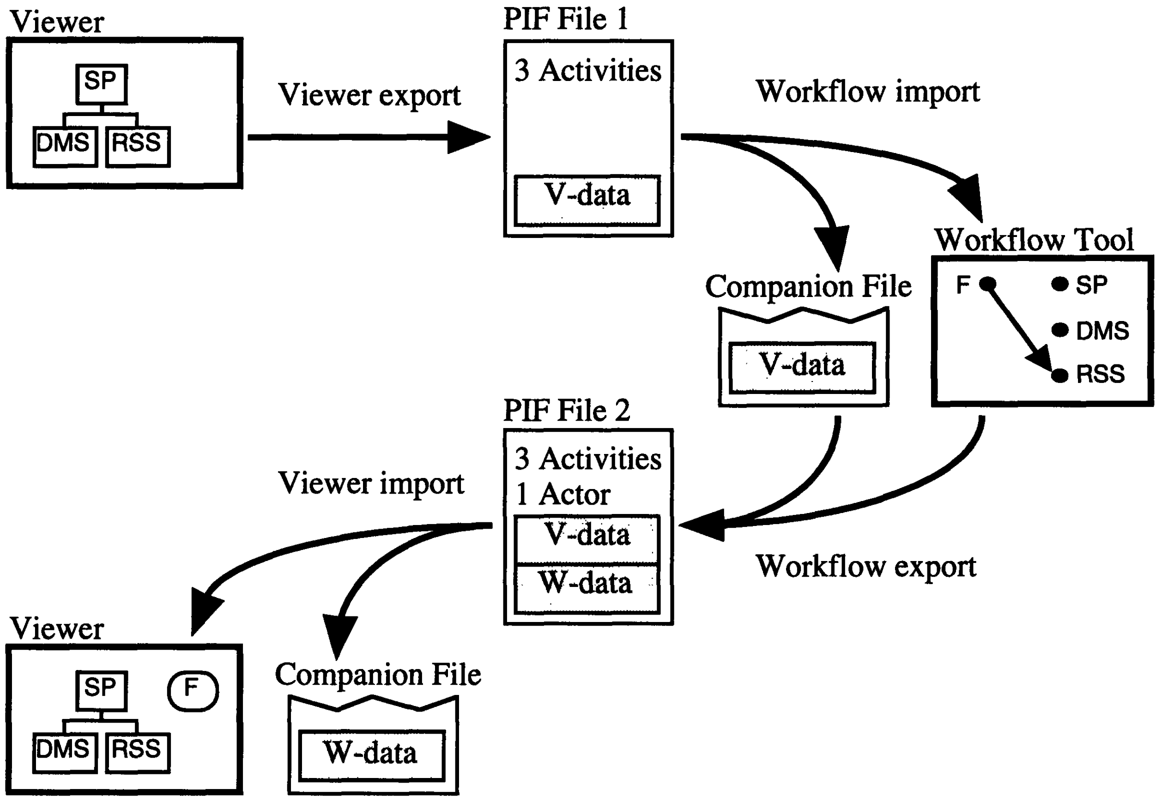

Figure 6: The solution for the earlier scenario.

3.2.2 The Genealogy Attribute

The solution to the second aspect of the problem, preserving type extensions, involves a new "Genealogy" attribute. We introduce this special attribute which documents the class information of the object before and after PSV conversion. Though discussed in more depth in section 3.3.3, it contains a list of types representing the path of conversion. Again analyzing the EMPLOYEE example (of Frame 4), there are two main possibilities when loading the EMPLOYEE object from system X into system Y. If system Y can handle arbitrary attributes reliably, the companion file is not necessary. For the solution in this case, the only new item of data required is a “Genealogy” attribute with the value (EMPLOYEE ACTOR). Thus, the converted ACTOR object will have three foreign attributes in the system: “Medical-Plan,” “Salary,” and “Genealogy.”

However, if system Y cannot accept them, or the user does not wish to burden it with them, the translator must employ the scheme diagrammed above. The converted ACTOR object represented by Frame 6 is stored in system Y. However, the three foreign attributes are now saved in the companion file as in Frame 7. Technically, there are two frames below with the same identifier, but the whole entity is considered to be the (disjoint) union of both sets of own-slots. Perhaps system Y can only write the user attributes in as a comment to the user, but the second frame holds the only official version. Between the system and the companion file, each piece of data must be represented exactly once.

```
; Frame 6: After the X-->PIF1-->Y translations, the data of the
;          PSV-converted ACTOR object represented in system Y.

(define-frame 720909171700FYC
      :own-slots (
            (instance-of      ACTOR)
            (Name             |Frank Chan|)
      )
)

; Frame 7: After the X-->PIF1-->Y translations, the data of the
;          converted object in the companion file.

(define-frame 720909171700FYC
      :own-slots (
            (User-Attribute    '(Genealogy       EMPLOYEE ACTOR)
                               '(Medical-Plan    "Prudential")
                               '(Salary          1.99))
      )
)
```

To begin the return trip, the intermediate system-Y export will be combined with the companion file to yield the complete export. The overall theme in this merge operation is that the companion file is subordinate to the system’s representation. Basically a task of version management, there are five cases of interest:

- If an attribute was created in system Y, like “E-Mail-Address,” it should be incorporated as in the merged example that follows (Frame 8). Of course, the attribute may have the same name as one already in the companion file, such as “Salary.” Perhaps, an ideal translator would alert the user and rename the older attribute. Otherwise, the newer value from the system should override the previous value.

- If an object was created in system Y, it will simply be included in the resultant PIF file. There will be no companion data for that object to merge.
- If an attribute was modified (i.e., the object was modified), this change will be displayed after the merge also. Not only was the attribute in the system a "live" copy – it was the only copy. (This is why each item of data is represented only once.) In Frame 8, we display such a modification by adding a middle initial to the "Name" field of the ACTOR.
- If an attribute was deleted in system Y, it will not appear in the complete export file. Similar to the previous case of modification, deletion is a necessary power given to the system by entrusting it with the only representation of the attribute.
- If an object is deleted in system Y, the object's user attributes in the companion file will be ignored. Because the system representation is the primary source of data, the translator will only seek to merge companion data for objects that still exist in the raw Y-export.[9] Looking back at the handbook example (of Figure 6), if "Direct Mail Sales" is deleted in the workflow tool, the activity's graphical layout information from the viewer should also be deleted in this merge operation.

```
; Frame 8: The object exported from system Y (Y-->PIF2 translation),
;          after the merge of the companion file.

(define-frame 720909171700FYC
      :own-slots (
            (instance-of        ACTOR)
            (Name               |Frank Y. Chan|)
            (User-Attribute     '(Genealogy          EMPLOYEE ACTOR)
                                '(Medical-Plan       "Prudential")
                                '(Salary             1.99)
                                '(E-Mail-Address     "mickey@mit.edu"))
      )
)
```

In the fourth and final stage, we look to reinstate the system-X class extensions of the ACTOR objects. Upon import into system X, the translator first checks for objects of types not in view, namely, objects of system-Y extensions not used in system X. As before, these objects will be converted to ancestor types according to the familiar PSV scheme. Among the objects of recognized types, the translator then searches for the

[9]The assumption made here is that every object was translated to system Y, even if only as an ENTITY. In practice, we may want to exclude or only include certain types of objects. Those objects not imported can be stowed away in the companion file and tagged as excluded on import and complete, so as not to give the appearance that they were deleted in system Y.

"Genealogy" attribute. Since the subtype EMPLOYEE is in view, we can restore the object's original type and reinterpret the user attributes in this context (Frame 9). In anticipation of a second export from system X, a second companion file preserves those attributes that are not part of the EMPLOYEE type (Frame 10).

```
; Frame 9: After the PIF2-->X translation, the data represented back in
;          system X.

(define-frame 720909171700FYC
      :own-slots (
            (instance-of        EMPLOYEE)
            (Name               |Frank Y. Chan|)
            (Medical-Plan       "Prudential")
            (Salary             1.99)
      )
)

; Frame 10: After the PIF2-->X translation, the data in the second
;           companion file.

(define-frame 720909171700FYC
      :own-slots (
            (User-Attribute    '(E-Mail-Address   "mickey@mit.edu"))
      )
)
```

3.3 Discussion

Upon further analysis of this solution framework, we answer some concerns and make a few additional requirements and assumptions.

3.3.1 Preserving Identity

First, each PIF object must have an identifier (ID) that uniquely identifies it across all systems so that other objects can refer to it with certainty. These object IDs should be assigned upon creation and preserved over time and translations. This is most necessary when merging exported objects with their "companion" objects. An additional stipulation is that they cannot be modified within a system. Otherwise, such a change will be interpreted as the deletion of the object with the old ID, followed by the creation of an object with a new ID.

Currently within the handbook, we are using what appears to be a satisfactory scheme (locally, at least) that consists of a time stamp (twelve digits in the format "yymmddhhmmss") concatenated with the user's initials.[10] Of course, the danger in just using initials is the possibility that two users with the same initials will create objects in the same second. Alternatives that are more likely to be universally unique include the full name, phone number, Internet e-mail address, or Internet host address. In any case, the time/user combination has the supplemental feature of providing some documentation about when the object was created, as well as where or by whom. Even with this user information, a more likely scenario for conflicts occurs when two people are both operating in a system environment configured for the same person, accidentally or intentionally. Thus, the underlying need is to capture the uniqueness either in each client that creates PIF objects or in each instance of code that generates IDs.

In some respects, the ID is just another attribute of the object, so this requirement can pose a problem if it cannot be stored. However, since it is a special attribute essential for reidentification, we take extra measures to preserve it within a system. In decreasing order of preference, here is the decision tree for choosing an importing translator method:

1. Can the target system use PIF IDs to identify objects internally? If yes, this is the ideal situation, as objects can be exported again with this ID. If no, proceed to step (2).
2. Can the target system store PIF IDs as a user attribute? If yes, this is almost as good, as the exporting translator will just have to find this attribute. If no, move to step (3).
3. Does the target system have its own method of identifying objects? This ID scheme must be at least locally unique, ensuring a one-to-one mapping between local IDs and PIF IDs. If yes, continue with step (4). If no, skip to step (5).
4. Does the target system allow the translator to access to this information? This may involve a "second pass" of the translator to go back and see what IDs were generated. If yes, each object's local ID is stored as a user attribute in the corresponding companion object, which has its own PIF ID. On export, we can reidentify objects by their local IDs but use the PIF IDs in the resulting file of frames. If no, the local ID is not useful since it is kept internal, so try step (5).
5. Can the target system can store PIF IDs as part of the name or some other attribute? If yes, the import proceeds with the assumption that they will be preserved. We can even append a checksum to the ID to guard against its corruption. When an object is

[10]In some cases, an additional counter (within each second) was necessary since more than one object could be created in a second, such as in a batch operation or the duplication of a subtree.

exported from the system, the translator will have to extract the ID from the attribute in which it was stored. If no, go to step (6).

6. What we have is a system that cannot even store one attribute value of reasonable length (i.e., long enough for a PIF ID) for each object. While other system-specific "work-arounds" may be possible, we give up on such an unlikely system for now. If there are no means for reidentifying an object after it passes through a system, there is no round trip solution in this case. Of course, the user will probably still want to import the file of objects, but there will be no guarantee of matching them with their companion objects on export.

For systems that do not use a globally unique ID scheme, an exporting translator should assign time/user PIF IDs to newly created objects or objects that do not have a PIF ID preserved in one of the measures above. If necessary, an object's local ID can be stored as a user attribute and reused when the object is reloaded back into the non-conforming system. The disadvantage of this approach is that the same set of system objects exported twice will have different PIF IDs. While it is possible to intelligently guess (perhaps based on their local IDs or name) if two such sets of exported objects came from the same set of system objects, one cannot say with certainty. This should provide an incentive for systems to have a scheme which stamps each object with a unique ID at creation.

3.3.2 Naming Conflicts

While systems that strive for ID uniqueness should not encounter conflicts, name clashes in type extensions and attribute names are more likely to occur. However, PIF's naming problem is not much different from others in information systems. The namespace of classes can be treated like that of procedures in any programming language. Parties who wish to share views of subclasses should come to an agreement on their names and meaning. This situation is reminiscent of the task of defining an interface between two code modules in software engineering. On the other hand, those that wish to just create system-specific subtypes can prefix their names with the initials of their system, as in PH-BUNDLE, for singularity.

As in the merge of a newly created attribute described earlier, there can also be name clashes at the slot level. While similar steps can be made for sharing specific fields, accidental conflicts are more likely due to the "catchall" nature of the "User-Attributes" slot. When two systems use an attribute name, like "Salary," with the same meaning in mind, the replacement of the older value with the newer one is generally the intention. Of course, systems that actually plan to perform calculations with the value should agree on its semantics (hourly, monthly, yearly, etc.) or rename the attribute to be more specific (e.g., "Monthly-Salary").

However, when there is one name for two entirely different meanings, perhaps a "Compass" slot of an activity in both navigational and drawing contexts, we have no choice but to rename one of them. On the flip side, a related problem occurs when there are two names with essentially the same meaning.[11] Even within our handbook, we use the terms "decomposition" and "subactivities" interchangeably. We will later introduce a way for PIF slots to have aliases (section 4.2.2).

3.3.3 Genealogy Issues

As stated earlier, the genealogy attribute records class information from PSV conversions. This consists of a list of types starting with the object's original, most specific class and ending with its current, most general class. This complete list permits translators to later reinstate an object to some subtype in view even if the most specific subtype is not in view. Just saving the most specific class would imply an "all or nothing" approach, only allowing a return to the original type.

To show the operations performed on this list of types, here is an example of how it evolves in translations across three systems. Suppose the first system has an ENGINEER class, a subtype of EMPLOYEE. In a second system with only the EMPLOYEE type, the translator creates the genealogy when an ENGINEER object is generalized. Then, in a third system where neither ENGINEER nor EMPLOYEE are in view, the translator appends to this attribute when the object is further generalized to an ACTOR. Later, if we can only reinstate the object to the EMPLOYEE type in the second system, a substring of the genealogy still applies. Finally, when the object has its original type back in the first system, there is no need for a genealogy.

```
ENGINEER (1):                                            ; original type
EMPLOYEE (2):  (Genealogy ENGINEER EMPLOYEE)             ; generalized once
ACTOR    (3):  (Genealogy ENGINEER EMPLOYEE ACTOR)       ; generalized again
EMPLOYEE (2):  (Genealogy ENGINEER EMPLOYEE)             ; specialized once
ENGINEER (1):                                            ; specialized again
```

[11]These two problems (same name with two meanings, two names with the same meaning) are the subject other research, such as MIT's Context Interchange Project (Goh et al 1994).

Though one might question whether storing the current class in the genealogy is necessary, this seeming redundancy is used to confirm the object's type on export. Suppose an engineer object is imported into the third system (from above) as an ACTOR with an appropriate genealogy attribute (either stored in the system or in a companion file). If the object's type is changed in that system to a different local extension of ACTOR, say LABORER, the object's genealogy becomes invalid. This is because the ENGINEER and EMPLOYEE types are no longer valid subclasses of the new type LABORER.

Therefore, if a converted object does not have the same type on export from a system as it did on import, its genealogy information must be discarded. In other words, the object's new type becomes its original type. Otherwise, if the genealogy was preserved, the object's type would be ambiguous in yet another (fourth) system that shared both the EMPLOYEE and LABORER extensions of ACTOR. This genealogy deletion applies not only to type extensions but also to core types. Suppose the type of a relation was changed from USES to MODIFIES in a system. Then, its genealogy attribute containing foreign extensions of USES would have to be deleted.

While an object's type is not required to be fixed at creation like its unique identifier, changes in type are not recommended. In this paper, we do not question if one system should have the ability to change the type of an object created in another system. We just assume that such a change, perhaps a legitimate correction, is in the best interests of all parties sharing the data. Even if they do occur, this framework still prevents the loss of attributes since the change in type does not preclude the merging of other companion data.

One possibility would be to store a genealogy for every system whereby importing translators would only use the genealogy of the destination system. Since we only know the name of the system which creates a PIF file on export, the genealogy would now be created by exporting translators. Suppose there is an X-->Y-->X translation scenario where system X has the EMPLOYEE extension and system Y uses the LABORER type. Upon export from system X, EMPLOYEE objects are converted to ACTORs through PSV, and this conversion is recorded in the "X-Genealogy" attribute. When in system Y, the type of these ACTOR objects can be freely changed to LABORER objects. On the second PIF export, the LABORERs are converted to ACTORs, and this is noted in the "Y-Genealogy." Then, when the objects return to system X, the EMPLOYEE type can be restored since the X-import translator looks for the genealogy of its system.

While multiple genealogies would allow type changes and prevent the loss of local subtypes upon return to a particular system, this plan conflicts with the goal of having type extensions for sharing among different systems. Suppose we now have an extended round trip X-->Y-->Z-->X, where system Z also has the EMPLOYEE extension. Then, EMPLOYEEs created in system X will only appear as ACTORs in system Z since the Z-import translator does not use the "X-Genealogy" attribute.

Generally speaking, the disadvantage of using a genealogy attribute is that it can create maintenance problems. By potentially having hierarchy fragments in every object, there can be significant redundancy within a PIF file. The many versions of type information will introduce some danger of inconsistency among the genealogies of different objects. If the class hierarchy of a system is changed, there will be many locations where these changes should appear. Suppose that the EMPLOYEE extension was created later and inserted between the two existing classes, ENGINEER and ACTOR. Then, all existing engineers which had been previously converted to ACTORs would have genealogies (ENGINEER ACTOR) that need to be updated to (ENGINEER EMPLOYEE ACTOR). Otherwise, they could not be specialized from ACTORs to EMPLOYEEs in a system with only the newer EMPLOYEE extension.

An alternative would be to retain the "define-hierarchy" structure after importing a PIF file. In this approach, genealogy information would only appear once in this class tree instead of having redundant hierarchy paths in converted objects. Unfortunately, this does not eliminate the need for some form of a genealogy attribute. Translators still need to know both an object's original (most specific) type and its converted type for the reasons discussed earlier. Also, objects would be less self-contained, as their genealogies would be dependent on the external hierarchy structure. Even when importing into systems that can handle arbitrary attributes of objects, the companion file would be necessary if only to store the hierarchy tree. As described later in section 4.3.1, the define-hierarchy structure is generated for PIF files based on the ontology of the exporting system. Similar to the merge of attributes from the exporting system and companion file, this approach would also require the merge of the two type hierarchies on export.

3.3.4 Multiple Imports and Exports

Earlier, the feasibility of the solution framework was explored in a simple four-translation scenario between two systems. In these translations, a single file of objects represented the entire state of the system. For systems designed to operate on large databases, this is impractical. We now look at more complex, yet more realistic, scenarios of exchange involving multiple imports and exports of PIF files.

When reimporting the same set of objects into a system, the translator must have a replacement strategy. That is, one must decide how a system should handle the import of data already present in some form. Though version management is certainly not the focus of this thesis, we outline three options: object replacement, slot replacement, and slot value addition. Besides not importing preexisting objects, the translator's most basic plan replaces the old object and its attributes with a new one from the imported PIF file. While this would purge the previous object and any accumulated attributes, any references in the system to it may have to be updated to the new object. A more sophisticated procedure keeps the existing system object and operates on each slot independently. Here, we only replace an object's attribute if the slot is defined in the new PIF object. Stepping one level deeper, a third scheme just adds attribute values if they were not already present in the object's slot. We leave the choice to the person writing the translator.[12]

When we import different sets of objects into a system, we have two options in dealing with multiple sets of companion data. We can maintain the status quo and leave a companion file on each import. Then, instead of just merging a single companion file, the exporting translator can draw system-specific data from as many companion files as the user provides. Unfortunately, this will be a burden for the user if he or she must keep track of all these files. Suppose a user imports five sets of objects and later decides to export one object from each of the five sets. In order for a translator to combine the data of foreign systems, it must have access to all five companion files. More importantly, while the companion data is clearly subsidiary to the system data, there is no such relationship between various companion files. We will either need a more complicated replacement strategy or a chronological ordering of the companion files.

Instead of postponing the merge of companion data until export, the alternative method would merge on import. By adding to a single companion file on each import, we have a chronological ordering on what would otherwise be multiple companion files. This allows the translator to enforce a replacement strategy when we again import those objects whose companion data has already been saved. In effect, we could have a single

[12]The real problem seems to be that the meaning of the absence of a slot has not been defined in PIF. It is not certain whether this reflects a null value or a lack of information. If an ACTOR instance does not have a name slot, for example, we are unsure if the actor has no name or if the name is just not known or available.

companion data repository for each database or system. The disadvantage of this approach is that this repository can become bloated with the data of foreign systems for every object ever imported into a system. It may be useful to occasionally perform some form of "garbage collection" on the repository for objects that have deleted from the system. A companion file should only be deleted when all its objects have been deleted or the user is sure that the system objects will never be exported again.

In section 3.3.1, we have already realized the ramifications of multiple exports from systems that only use local, non-unique IDs. If new PIF IDs are created on each export, it will be difficult to recognize that they are actually the same objects.

When we import the same set of objects into multiple systems, we encounter a whole new set of potential problems. Suppose a system X exports an EMPLOYEE object (as in Frame 4) for import by system Y and Z. A user in system Y then changes the employee's "Medical-Plan," while another user in system Z raises the employee's "Salary." When the two resultant PIF files are imported back into system X, only one update will persist if all fields are updated according to one of the replacement strategies above. Despite the need for some form of concurrency control, we do not deal with this problem here. Replication issues in distributed databases have been given much more thought elsewhere. For the purposes of this thesis, if we can prevent system-specific data from being lost, we have accomplished our goal. Without this solution, there may not be "Salary" and "Medical-Plan" attributes to update.

3.3.5 Practical Considerations

One can foresee that in a longer round trip, say V to W to X to Y to Z back to V, the resulting PIF and companion files will contain a great deal of "baggage," mainly user attributes for as many systems with which the PIF file comes in contact. When importing this file into a particular system, the companion file may have to contain all data from other systems accumulated from previous translations. However, the user can always choose to start "fresh" by either declining to leave a companion file on import or ignoring it on the ensuing export. Though not discussed here, this raises an interesting issue about what incentive does a PIF user have to preserve data from other systems. Nevertheless, when one party is using multiple systems on the same set of objects, the benefits are more clear. The user will not have to manually merge system-specific data that would otherwise be lost in translations.

On a related note, we will not view storage space as a major concern. This approach to the problem might not be well suited for the interchange of large picture files, where file size and compression are crucial. Given our focus on the handbook, this currently does not appear to be an issue since exchanged files have generally been text-based. On the other hand, this solution should encourage the inclusion of other forms of data. If PIF users know that these forms of data can be preserved in a round trip, they will be more inclined to include them in their descriptions. We hope this leads to the exchange of richer business process descriptions.

4 The PIF Toolkit

4.1 Purpose

We now look to build translators that employ this solution framework, and the PIF toolkit is the foundation of all such translators. Development on this code library began as an outgrowth of writing the first PIF translators for a couple of handbook prototypes. As one might have guessed, we first encountered the round trip problem when sharing process descriptions between these two prototypes. A simpler solution suited just for this pair could have been developed. Perhaps we could have even modified the systems to be more similar or to accommodate the data of the other. However, we sought a more systematic way to preserve data in exchanges, not only between other handbook tools but also non-handbook systems.

In fact, solving the problem at hand was not the initial goal of the toolkit. Our first motivation was to consolidate the common elements of all current translators, making it easier to accommodate changes either in the PIF ontology itself or the tools that use PIF files. From conception to the release of version 1.0, PIF has undergone changes at all levels, from the meanings of various classes all the way down to the smallest of syntactical details. Moreover, system architects have worked and will continue to work on the tools that read and write PIF files even if the format were completely frozen. We would like for changes in either domain to require as little modification as possible of past translators.

In the larger scope of the PIF working group, we have also performed experimental translations with the systems developed by the other research groups. Having a common code base to encompass all the PIF data structure manipulations, such as parsing and output, saves on the collective development effort of writing translators. Besides smoothing the standardization effort (for example, ensuring the same syntax), this permits members to focus more on furthering the format. Outside the working group, we hope that this toolkit will promote the sharing of process data through PIF, helping to establish PIF as a standard. It will simplify the task of writing future translators, not only for our group but also for anyone who would like to share business process descriptions. Even if the toolkit did not incorporate a round trip solution, it would still be beneficial to our project. Each additional system that can export PIF files brings a world of process descriptions to the handbook.

Finally, we are obligated to define the scope of the toolkit. First, just as PIF is not meant as a replacement for KIF, this is not a substitute for more general knowledge representation tools. It is designed specifically for building PIF translators, as opposed to a general-purpose development environment or programming language like Common Lisp. Second, this toolkit does not operate on the semantics of PIF entities like an inference engine. For example, in addition to "own-slots" and "template-slots," Ontolingua also gives frames "axioms," which are KIF sentences to say things that cannot be expressed in terms of the slots (Gruber 1993). Axioms could be used to express invariants between attributes as first encountered in section 3.2. Besides the additional code complexity such analysis would demand, there are currently no formal semantics for (or constraints on the interpretation of) PIF objects. Thus, one can express an ACTIVITY that is a "Component" of itself or the pair of objects PREREQUISITE(A, B) and PREREQUISITE(B, A), for two activities A and B.[13] Other than type checking, this toolkit does not pretend to detect these seemingly illogical constructs.

4.2 Toolkit Design

4.2.1 Building Blocks

The objective of translation was the primary factor in the choice of the implementation environment. Considering the relationship between KIF and Lisp and the existence of systems like Ontolingua, it might seem that the natural choice of programming language would be Common Lisp (perhaps with CLOS, the Common Lisp Object System). However, for the toolkit to be usable by as wide an audience as possible, we had to choose a language that would interface smoothly with systems in both academic research and commercial environments. Currently, "C" (Kernighan and Ritchie 1988) is the most widely accepted and most easily accessible software development environment. Some systems, like Lotus Notes, have an API (application program/programmer interface) only available in C. We would like to avoid the often tricky task of linking code of different languages together. However, we still sought some of the features of a higher-level language, namely ease of expression. We settled on C++ (Stroustrup 1991), an object-oriented superset of C, as a suitable middle ground. Though the use of C++ is not as widespread as that of C, it is becoming almost as accessible. We shall see later in section 4.2.2 that C++ is not as sophisticated as CLOS in some ways.

[13]In the latter case, this freedom of expression may be useful in describing a process in deadlock.

After the choice of language, we had to determine the toolkit's implementation target. As a building block for translators, the toolkit itself is not an executable program and has no graphical user interface. Rather, it is a code resource of generic C++ for maximum compatibility. More specifically, it is implemented as a platform-independent library of C++ classes. The C++ notion of "class" refers to a programming language object, not to be confused with a class in the ontology hierarchy.[14] Figure 7 displays the relationships between the building blocks of the PIF specification and the toolkit implementation. The toolkit also makes use of lex, a lexical analyzer generator, and yacc, a parser generator, which both generate C code.[15] Later, we will see how the specification of PIF and its extensions are inputs to the toolkit, as shown by the arrows.

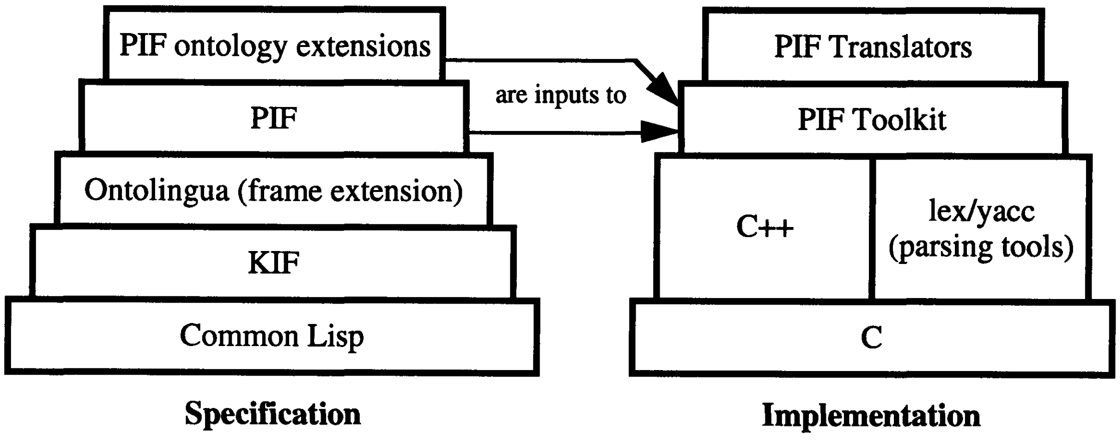

Figure 7: The building blocks of PIF and the toolkit.

4.2.2 The Metaclass Mechanism

The first translators for the handbook were built upon a set of classes which modeled the entities in PIF. In other words, there was originally a one-to-one correspondence between ontology classes and C++ classes. This design works best for an object model that is relatively fixed, such as that of a particular project. One can assign special methods to different objects. Unfortunately, it is not well suited for the PIF translation, where the main focus is on input and output. Unlike CLOS, which can create

[14]For those readers not familiar with C++ but proficient with C or other procedural programming languages, a C++ "class" consists of an object's data structure and associated methods. More primitively, this kind of class can just be thought of as a set of related procedures. To avoid confusion between the two kind of "classes" discussed in this paper, we will refer to the earlier kind as ontology or PIF classes and to the later kind as C++ or code classes.

[15]Actually, the GNU Project's flex and bison, respectively, were used. These versions are available from the Free Software Foundation via anonymous ftp at prep.ai.mit.edu.

classes dynamically or at "run-time," C++ classes are static or frozen at "compile-time." This would mean that changes or extensions to the PIF ontology would require recompilation of the toolkit due to code modifications of the corresponding C++ classes.

To overcome this serious limitation of previous translators, this toolkit has a metaclass mechanism. Booch defines a metaclass as "a class who instances are themselves classes" (Booch 1994). This definition fits our design perfectly because instances of one of the toolkit's C++ classes (naturally named "Class") represent PIF classes. The toolkit does not model the particular classes of the PIF ontology but has the capacity to describe any ontology class. With meta-objects, we can reproduce the previously mentioned ability of CLOS here in C++.

Though more complicated, this design allows us to dynamically load the PIF class hierarchy as well as any subtypes. The arrows in Figure 7 represent how the ontology and its extensions are now considered inputs to the toolkit. Besides the added complexity, the main drawback is that we sacrifice the ability to attach special methods to a particular PIF class. By abstracting away from specific object types, though, we treat them much more uniformly. Hence, the toolkit and the translation mechanisms will also function on a hierarchy of message types, as in the PSV paper (Lee and Malone 1990), or even animals.

In order to set the class hierarchy dynamically, the toolkit will load one or more ontology files, only containing PIF classes (like Frame 1), as a secondary input. We are still not introducing classes into the format itself, as these class files are loaded separately from the instance files, the primary input. PIF files themselves still only contain instances. To specify either the set of core types or type extensions, frame syntax is used again. As discussed earlier, classes have template-slots as well as own-slots, unlike instances.[16] In this toolkit, template-slots can only contain facets.[17] A facet describes a property or constraint on all instances of a given class. Each slot can have any number of facets which together define the type of values allowed. Here are the five facets recognized by the toolkit – the first four come from Ontolingua, the last one was created for the toolkit.

[16]In the toolkit, we only make use of two types of own-slots for classes: "instance-of," which is always set to CLASS, and "subclass-of," which refers to the parent class. In addition, the "documentation" slot may be useful. Ontolingua recognizes many other own-slots.

[17]More generally, template-slots of classes can also contain values. These become default values of instances of the particular class. In other words, any such values are downward inherited to all instances. This has the same effect as the inherited-slot-value facet introduced by Ontolingua but not supported here.

- (**slot-value-type** ?type) – Slot values must be of type ?type, where ?type is one of the following: a literal data type, the symbol CLASS, or a symbol specifying a class. If a slot has a (slot-type-value CLASS) facet (e.g., the core PIF class GROUP), the slot values must be classes. Otherwise, and more commonly, the slot values must be instances.
- (**slot-cardinality** ?integer) – The number of slot values must be exactly ?integer.
- (**minimum-slot-cardinality** ?integer) – The number of slot values has the lower bound ?integer.
- (**maximum-slot-cardinality** ?integer) – The number of slot values has the upper bound ?integer.
- (**slot-alias** ?symbol) – The slot is also recognized under the name ?symbol for input. However, on output, the actual slot name is used.

The following example (Frame 11) displays the EMPLOYEE class definition that would be used by a translator constructed with the PIF toolkit. In the example of section 3.2.2, this information will be used when the EMPLOYEE object is reinstated back to its original type in system X. Given this definition, the translator will know which attributes are a real part of the type extension (as in Frame 9) and which are just user attributes (as in Frame 10). Each attribute has a facet with applies a constraint on the type of values which can appear in EMPLOYEE objects. There is also a second facet, a cardinality restriction, for the "Salary" attribute because it only makes sense for an employee to have one salary.

```
; Frame 11: The frame for the EMPLOYEE class extension.

(define-frame EMPLOYEE
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ACTOR)
      )
      :template-slots (
            (Medical-Plan     (slot-value-type  STRING))
            (Salary           (slot-value-type  NUMBER)
                              (maximum-slot-cardinality     1))
      )
)
```

A subtype can redefine and thus hide the slot of its parent type, so only the definition of the most specific class matters to an instance. Thus, facets such as a cardinality restriction do not "accumulate" and should reappear in the template-slot of the child. Redefining slots is usually a way of specifying stronger constraints on slot values. For example, both ENTITY and ACTIVITY in PIF define the "Component" slot to be of their own type. This means that entities can be composed of all entity types, but activities can only be composed of activities (or instances of ACTIVITY subclasses). However, for any given class, the PIF toolkit does not currently enforce any rules on what kinds of facets can appear in its subclasses.

4.2.3 Representing Frames in C++

After specifying the type hierarchy (classes) to accompany our process description (instances), these two collections of PIF frames need to be represented by C++ objects in preparation for translation. Figure 8 displays the code classes that model ontologies and PIF files in a table of "what of where" relationships. For example, attributes of actual PIF objects are modeled by the C++ class OwnSlot. Each row represents a level of aggregation such that any code object is composed of the code objects below it. Each column represents a different context in which the code objects are used. The abstract column contains base code objects from which objects in the other two columns inherit (in the object-oriented programming sense). Though these twelve are the most important, they are by no means the only classes in the toolkit. Others include iterator constructs and more specialized classes for the different kinds of slot values and facets.

where / what	abstract objects	PIF object types	actual PIF objects
object collections	**Collection**	**ClassColl** (i.e., ontology)	**InstanceColl** (i.e., PIF file)
objects	**Frame**	**Class**	**Instance**
attributes	**Slot**	**TemplateSlot** **OwnSlot**	**OwnSlot**
attribute values	**SlotItem**	**SlotFacet**	**SlotValue**

Figure 8: The C++ classes of the toolkit.

Though the actual code is not discussed much here, member functions (methods) of the code objects fall into four categories:

- constructors/destructors: These perform the allocation and deallocation of memory, as well as the initialization of the object's data fields.
- observers: Procedures that do not modify the objects may be as basic as getting the name of a slot. They can also be more elaborate, such as those involving type-checking (i.e., verifying that an own-slot matches its corresponding template-slot).
- mutators: Similarly, there are simple and complex operations which do modify the objects. They range from setting a slot name to converting the type of an object to its nearest parent in view.
- input/output functions: The most significant of these involves the ability of each object, from collections to slot values, to be able to print itself in the PIF syntax.

4.3 Translation Algorithms

With an internal object representation for the solution framework, we are now ready to present a more formal description of how the toolkit exports and imports PIF files. Steps in these algorithms are performed by some of the objects' more complicated member functions, which are, in turn, built upon the elementary ones. Both have four general phases as shown in Figure 9: (1) input of the object "dictionary" (classes), (2) input of objects to translate (instances), (3) processing, (4) output of translated objects (also instances). The processing phase is where most of the tasks associated with solving the round trip problem occur.

Besides these tasks for the round trip solution, the second novelty of the toolkit is the aforementioned metaclass mechanism. By dynamically loading the type information in the first phase, translators written with this toolkit become "programmable" since their behavior can be changed with different sets of classes. Each PIF translator written with this toolkit will load a file containing the core ontology described in section 2.2.2 for basic functionality. Similar to computer hardware peripherals, sets of type extensions then act as "plug-in" cards or modules by which one can customize a translator.

Though the solution framework makes use of partially shared views, one may find it useful to think of the return trip procedure as the inverse operation of the PSV scheme. The PSV scheme generalizes objects to more generic supertypes, but the toolkit also specializes them back to their more specific subtypes when they are again in view. PIF's "define-hierarchy" structure is used in the former, while genealogy attributes and ontology descriptions allow the latter. At the same time, an object's attributes can be demoted to the user-attribute repository and later promoted back to their original status.

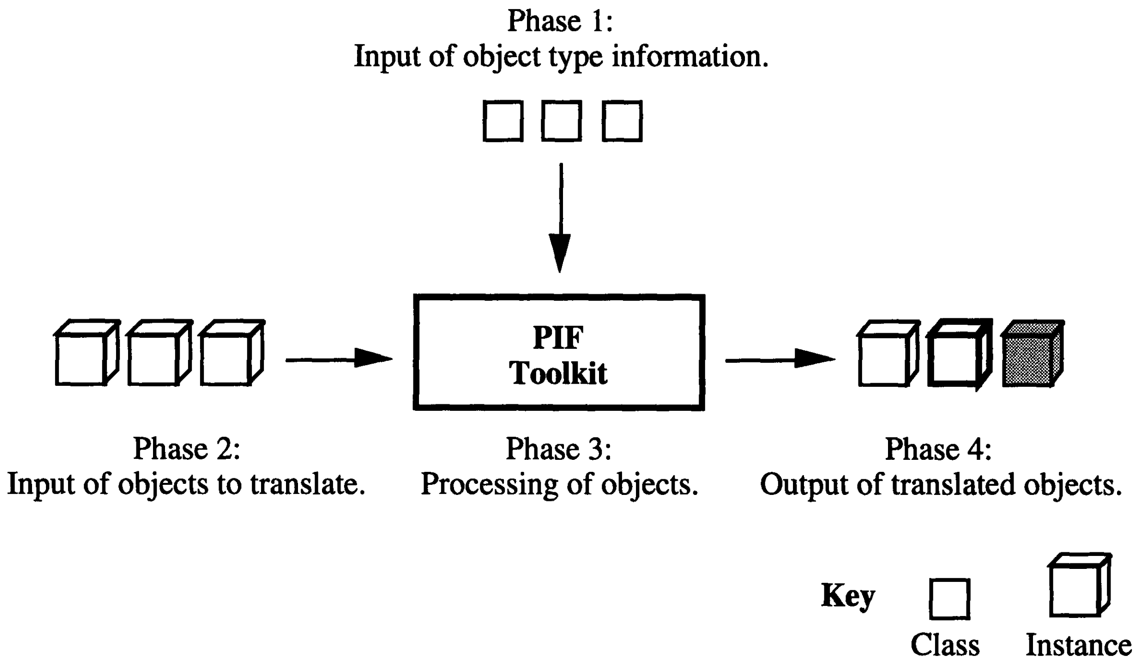
Phase 1:
Input of object type information.
PIF Toolkit
Phase 2:
Input of objects to translate.
Phase 3:
Processing of objects.
Phase 4:
Output of translated objects.
Key
Class
Instance

Figure 9: The four phases in PIF toolkit translations.

4.3.1 Export Algorithm

The PIF export is the simpler of the two operations because we are only working with one class hierarchy. This working ontology will reflect types used in the exporting system, the source of the objects. Therefore, all objects will have the correct type for this context, so there will be no need to specialize or generalize any of their types. The main "highlight" is the merge of data from the system and the companion file. If the merge is not necessary, perhaps because the objects were just created, this operation merely becomes an export of the system's state.

For PIF translators, the first stage will usually entail first loading the core PIF classes and then any extensions used by the exporting system. However, the toolkit itself and its metaclass mechanism are neither bound to the hierarchy of PIF nor a single set of extensions. There may be multiple roots and multiple levels of extensions. The files of classes are loaded sequentially, though, so each file of classes must be defined in terms of itself and any types previously loaded. Thus, the first set of classes must be completely self-contained.

Export Algorithm Stage 1: Building the source ontology (ClassColl).

```
FOR each file:
    • Parse frames (using lex/yacc tools).
    • Create (C++) Class objects and add them to ontology.
    FOR each class:
        • Check values of own-slots.[18]
            • The "instance-of" slot must be the symbol CLASS.
                                        {I.e., frame is a class, not an instance.}
            • If it exists, the "subclass-of" slot must contain a class in the ontology.
                        {We can then build the hierarchy by linking this class with its parent.
                                  If the slot does not exist, this class is a hierarchy root.}
            • If it exists, the "documentation" slot must be a string.
        FOR each template-slot:
            FOR each facet:
                • Apply facet, making sure template-slot is not over-constrained.
    • Make sure there are no cycles in the hierarchy.
```

Though a little extra effort is sometimes necessary to preserve PIF IDs, the second phase reduces to a loop over the objects and a nested loop over the object's attributes, setting the ID and type in the first and the attribute values in the second. After this, we have an intermediate collection representing the state of the system. Obviously, the questions concerning the use of local IDs instead of PIF IDs do not need to be asked every time an entity is translated. These decisions will probably be made by the translator writer (i.e., at "compile-time").

Export Algorithm Stage 2: Building the collection of objects (InstanceColl).

```
FOR each object in the system:
    • Create an (C++) Instance and add it to collection.
        IF (system object has a PIF ID) THEN
            • Create an Instance with this PIF ID.
                                {the ID having been generated at object creation or
                                 preserved by any of the means discussed earlier}
        ELSE
            • Create an Instance with a newly generated time/user PIF ID.
                                      {This includes objects with only a local ID.}
        • Type must be a Class in the ontology (ClassColl).
    FOR each of the object's attributes:
        IF (attribute is defined by the Class or its parents) THEN
            • Set attribute, making sure value complies (type checks) with facets.
                          {Only use facets in attribute definition of most specific Class.}
        ELSE
            • Place attribute in User-Attribute repository.
    • If the object uses a local ID instead of a PIF ID, add it as another user attribute.
```

[18]Whenever something is checked, we assume for now that an error will be reported if the check fails.

The third stage concerns the merging of companion data from any number of companion files. Here, we assume that they are read in chronological order of modification. Thus, the newest companion file should have the latest system-specific data. This phase requires an extra step if only local IDs could be stored in the system on import. Namely, we have to reinstate the PIF ID of objects which could only be reidentified by local IDs. Now, companion attributes and objects have been merged with the intermediate collection to yield a complete collection. As determined in section 3.3.3, the genealogy attribute is discarded if the object's type on export is not the same as its type on import. This type comparison is performed after the merge because the genealogy attribute may be stored in the companion file.

Export Algorithm Stage 3: Merging the companion data (another InstanceColl).

```
IF (there are any companion files) THEN
    FOR each companion file:
        • Parse frames and add them to companion collection.
    FOR each Instance in intermediate collection:
        IF (there exists a companion object with the same local ID value) THEN
            • Change ID of intermediate object to PIF ID of companion object.
                                                    {in preparation for the next step}
        IF (there is a companion object with the same PIF ID) THEN
            FOR all attributes in companion object:
                • Add attribute to intermediate object if not already present.
                                                {Attributes from the system have precedence.}
    FOR each Instance in companion collection:
        • If marked as "excluded on import," add Instance to intermediate collection.
FOR each Instance in the merged collection:
    IF (the genealogy attribute exists) THEN
        IF (current type does not equal previous type from genealogy) THEN
            • Delete invalid genealogy attribute.
```

At first glance, it may seem possible to specialize objects with genealogies back to their original types on export. For example, in the export from system Y in Frame 8, the ACTOR object could conceivably be reinstated to an EMPLOYEE. However, this step is postponed until the PIF file is imported because the full class information of genealogy types is not available. With only the ontology of the exporting system, all that is known about the foreign EMPLOYEE type is that it is a subtype of ACTOR. In order to fully specialize the PSV-converted ACTOR in Frame 8 back to an EMPLOYEE, the translator must have the class definition (of Frame 11) to determine which user attributes can be promoted back to object attributes (as in Frame 9). To have an EMPLOYEE frame with the "Medical-Plan" and "Salary" fields as user attributes would be inconsistent.

In the last stage, we must generate the class hierarchy from the source ontology. This will allow other systems to use the PSV scheme on any local extensions of this system.

Export Algorithm Stage 4: Writing the complete collection of objects to PIF.

```
• From the source ontology, generate and print define-hierarchy structure.
FOR each object in the merged collection:
    • Print frame.
```

4.3.2 Import Algorithm

The import is the more complicated operation because it involves two class hierarchies: the ontology in which the PIF file was created and the ontology of the system into which we are trying to import. Though one might think that the import involves mapping the first to the second, such a class to class assignment is not necessary. More conveniently, one can understand this task as interpreting a collection of instances in the context of the destination ontology.

It may be useful to think of an ontology as a colored glass through which a system views a set of objects. When seen through different colors of glasses (class hierarchies), objects of various colors (types) can appear differently. In Figure 10, there are two sets of three objects and four importing systems with different ontologies (represented by the tags). To the omniscient observer, there is also a simple class hierarchy consisting of an object type, A, and two subtypes, B and C. In the first set, objects of different colors may be interpreted as of the same type if the system cannot see the difference. Those that have been generalized on import are shown dimmed. In the second set, objects of seemingly the same color can be distinguished by systems which recognize the specialized types in the genealogy attributes. Those that have been specialized are shown in bold.

Import Algorithm Stage 1: Building the destination ontology (ClassColl).

{same as export algorithm stage 1 except that class definitions of the target system are used}

Import Algorithm Stage 2: Building the collection of objects (InstanceColl).

```
• Parse define-hierarchy structure of input PIF file.
• Make sure there are no cycles in the hierarchy.
• Parse frames and add them to collection.
FOR each instance:
    • Make sure "instance-of" own-slot names a type present in
        either the source's define-hierarchy or the destination ontology.
                        {Otherwise, this is an error – a type unknown to both worlds.}
```

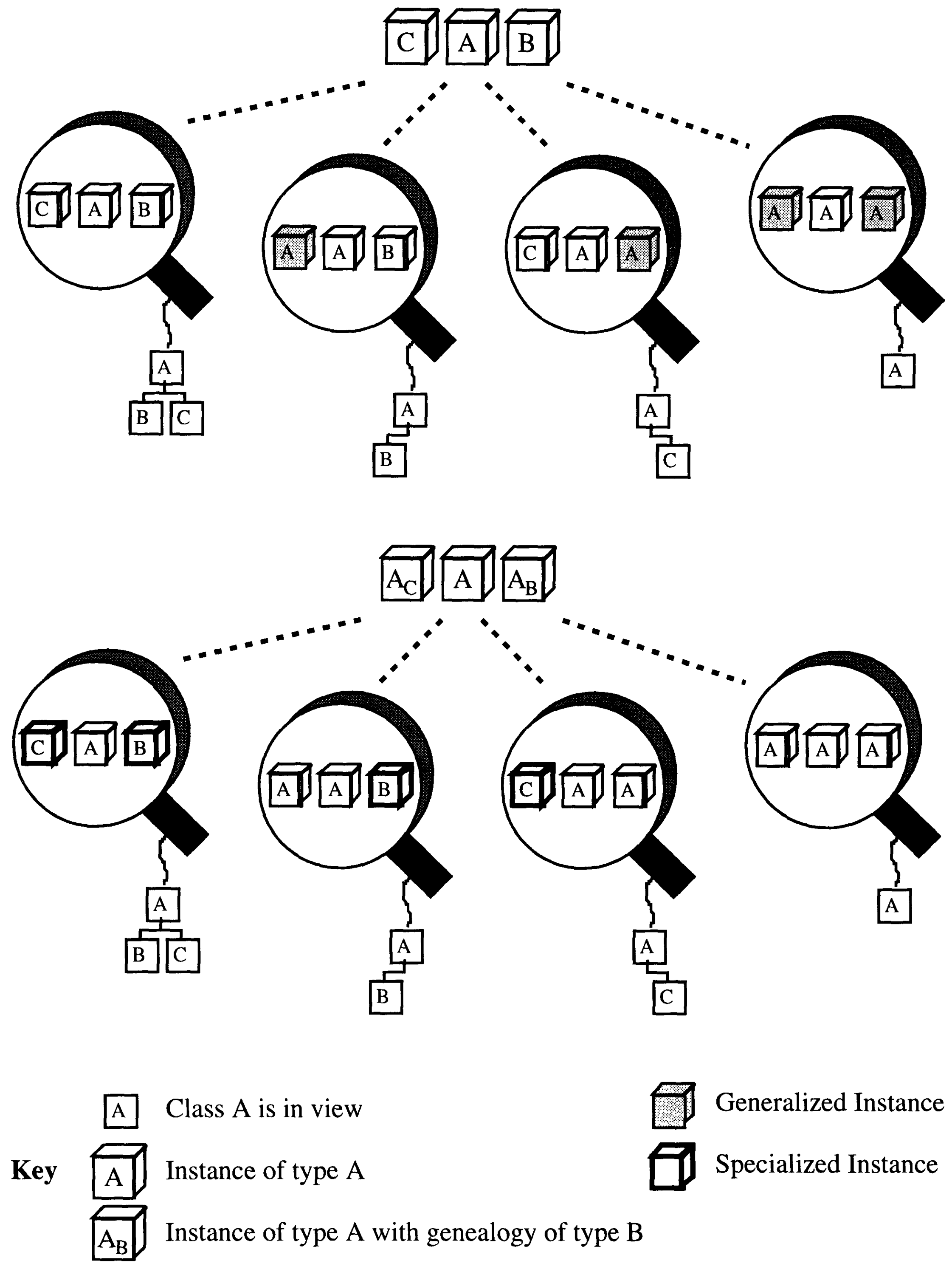

Figure 10: Interpreting instances through different class ontologies.

While the two input phases are rather routine, the third phase is where an object's types and attributes are reinterpreted. Here, a class is "in view" if it exists in the destination ontology. In order to specialize an object to a more specific subtype in view, we look to the genealogy attribute. On the other hand, if we must generalize the object to a supertype in view, we navigate through the source hierarchy. In both cases, we try to aggressively specialize or reluctantly generalize the object to the most extended type possible. Given the new type, the second half of this phase calls for the reinterpretation of all attributes. Both object and user attributes can be relocated based on the slot definitions of the appropriate class and any superclasses.

Import Algorithm Stage 3: Promoting and Demoting Types and Attributes.

```
FOR each Instance in collection:
    A. Types.
    IF (instance type is in destination ontology) THEN
        IF (instance has genealogy attribute) THEN
            IF (any subtype from genealogy is in destination ontology) THEN
                • Specialize instance type to most specific subtype in view.
                IF (new type is also the most specific subtype in genealogy) THEN
                    • Delete genealogy attribute.           {The instance is of its original type.}
                ELSE
                    • Set new genealogy to be the relevant substring of old genealogy.
            ELSE
                • Leave as is.                              {The genealogy types are not visible.}
        ELSE
            • Leave as is.                                  {The instance is of its original type.}
    ELSE
        • Using source hierarchy, find nearest ancestor type in destination ontology.
        • Generalize instance type to most specific supertype in view.
        IF (instance already has genealogy attribute) THEN
            • Append path of demotion to the old genealogy.
        ELSE
            • Write path of demotion to a new genealogy attribute.
    B. Attributes.
    FOR each attribute:
        IF (attribute is defined by the Class or its parents) THEN
            • Check type of attribute value.
                            {I.e., verify that the own-slot adheres to the template-slot facets}
        ELSE
            • Demote to user-attributes repository.
    FOR each user attribute:
        IF (attribute is defined by the Class or its parents) THEN
            • Promote to object attribute and check type of value.
        ELSE
            • Leave as is.                                  {The user attribute belongs there.}
```

After the semantic mapping of the instances into the local ontology, the fourth and final phase comprises of the syntactic mapping of these objects into the local representation. Similar to the export algorithm, there are additional steps here for systems that cannot use or easily store PIF IDs. Besides that, the main issue concerns if and where user attributes are stored. A desire to preserve round trip data may or may not call for leaving a companion file. When writing to a companion file, we can create a new one or add to an existing one, replacing old data if necessary. The toolkit provides the functionality to do both, but this decision is left to the translator writer.

Import Algorithm Stage 4: Loading the converted objects into the target system.

```
• If desired or necessary, designate certain types of objects as excluded from import.
FOR all other Instances:
    • Create/Update object in system (depending on replacement strategy).
                                    {preserving the PIF ID by any of the means discussed earlier}
    IF (PIF ID could not be stored and system has a local ID scheme) THEN
        • Save local ID as another user attribute in the instance collection.
    • Load/Update object attributes into system.
                                    {If necessary and possible, store PIF ID as part of an attribute.}
    IF (user wishes to preserve round trip data) THEN
        IF (system can handle arbitrary attributes) AND
                (user wishes to load them into the system) THEN
            • Save user attributes in the system.
        ELSE
            • Write user attributes to companion file.
    ELSE
        • Ignore user attributes.
FOR each excluded object:
    IF (user wishes to preserve round trip data) THEN
        • Mark object as "excluded on import" and write to companion file.
```

Similar to stage 2 of the export algorithm, the capability of the system to store PIF IDs and arbitrary attributes will be known to the translator writer. Thus, the actual algorithm performed at the time of translation (at "run-time") will be simpler.

5 PIF Translators

To demonstrate and test our solution framework, we must exchange PIF files between various systems in a series of round trips. In this thesis, the systems of primary interest are the various Process Handbook prototypes. Besides the static hierarchy limitation mentioned earlier (in section 4.2.2), previous handbook translators did not make any attempt to save the data of other prototypes. Now with the PIF toolkit, we can build better translators that preserve data specific to other systems. Based on the ontology input file of the target system, an importing translator can automatically determine which information cannot be interpreted. If this data is preserved internally or saved externally, an exporting translator can reassemble the system and companion data to yield a complete PIF file.

5.1 Two Steps to Build a Translator

To build a translator with this toolkit for any particular system, the first task is to define the system's ontology file(s). This means determining what types of entities and attributes of these entities are represented by the system in question. Though unlikely, it is possible that the set of core PIF objects and their attributes will be adequate. In this case, no additional work is necessary, as the translator writer can begin with the PIF 1.0 ontology file (in the appendix). More likely, each system will require type extensions, especially given the PIF working group's goal of having a minimal set of constructs. At the very least, systems will assign additional attributes to core classes.[19]

This difference between adding attributes to a core type and creating a new subtype with these attributes is subtle. Though the choice is important, it is certainly not irreversible. For any given core class, extensions are necessary when one has to make a distinction between different kinds of instances. Within a particular system, there may be multiple kinds of a core type, such as employee and visitor ACTOR objects. If so, this system ontology should contain the ACTOR subtypes EMPLOYEE and VISITOR. Then, after exporting these specialized objects, the importing translator can distinguish between

[19]The intention of the PIF working group was to not allow modifications to core classes. For our purposes, current and future versions of the core ontology should be published in frame syntax, and toolkit users will then be free to customize their own copy. Any such local changes will not reflect the official ontology, of course.

the two. Otherwise, if employee and visitor objects were only exported as ACTORs without any differentiation in type, this would constitute a loss of type.

Among two or more systems, new classes should also be created when sharing objects of more specialized types. The interested parties should jointly arrive at the definitions of these types for what will act as a supplemental type module for their translators. For example, the PRODUCER-CONSUMER-DEPENDENCY class is a relation type used in most handbook tools. If these dependencies were just exported as RELATION objects, import translators for these tools could not treat them as dependencies.

On the other hand, if there is no need for distinction, a system should not create a new type. There would be no additional benefit over adding attributes to a core class. As an example, handbook translators continue to use the ACTIVITY class instead creating a type like PH-ACTIVITY. The motivation behind creating a new subtype should be to denote a genuine difference of meaning from the supertype, not just a difference of system environment. While doing so anyway does not create a problem on export of PH-ACTIVITY objects, the real problem occurs on import. If we specialize another system's ACTIVITY objects to PH-ACTIVITY objects for internal use, we will have to discard any genealogy information which will likely include more significant subtypes (recall section 3.3.3). As a rule of thumb, if every object of the supertype would end up being specialized in a system to the new subtype, the subtype should not be created. Referring to the EMPLOYEE class defined in Frame 11, if every actor in a system was an employee, this new class would be unnecessary. The importing translator could use following customization of the ACTOR type (Frame 12) and still know which attributes are specific to the target system.

```
; Frame 12: A system's customization of the ACTOR type.
;           The Medical-Plan and Salary attributes have been added.

(define-frame ACTOR
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ENTITY)
      )
      :template-slots (
            (Has-Skill        (slot-value-type SKILL))
            (Medical-Plan     (slot-value-type STRING))
            (Salary           (slot-value-type NUMBER)
                              (maximum-slot-cardinality 1))
      )
)
```

The second task in building a translator is writing the code to extract or load PIF objects. This corresponds to export algorithm stage 2 and import algorithm stage 4 in sections 4.3.1 and 4.3.2, respectively. The other stages do not require additional work by the translator writer. At a superficial level, these two stages have a loop to iterate over all entities and a nested loop to iterate over all attributes of each entity. In some systems, it was necessary to create an object before referencing it in another object's attribute. Import translators for these systems required a two-pass algorithm whereby objects were translated on the first and their attributes (and references) were translated on the second.

In dealing with the actual objects of the system, there are two general approaches. If the system has its own file format, translators will perform a mapping between that and PIF. This may involve the use of existing utilities for parsing and writing to the native system format. If the system does not have a format or the format is somehow inaccessible (perhaps proprietary), it will be necessary to call toolkit functions within the system or a system application. Even with access to a native format, it may be simpler to take the latter approach.

Once the issues of mapping a native system representation to PIF are resolved, most of the work to read and write objects and attributes is mechanical and not of much theoretical interest. In our experience, the primary lesson to acknowledge is that while this solution framework does preserve system-specific information on a round trip, the task to build back the system representation from textual attributes is left to the translator writer. For example, the translators for the Lotus Notes prototypes must encode binary data as pure text upon export, and this text must be decoded back on import. On a related note, we found that it was better to export the minimum amount of system-specific information necessary to restore state, even if it required more processing on import. The presence of redundant information can lead to inconsistency when another system modifies only one appearance (as first seen in the "Name" and "Initials" example of section 3.2).

In any case, most issues in this task of translator implementation are specific to the system. We illustrate some of the issues possible by studying those encountered in writing translators for the handbook.

5.2 Translators for the Process Handbook[20]

5.2.1 Handbook Ontology Extensions

In building translators for the various prototypes with the PIF toolkit, the first step was to define a set of type extensions that fully represent handbook data. These are found in the appendix. As established in the project overview (section 2.1.1), bundles and dependencies are an important part of the handbook's descriptions of business processes. While the bundle only requires a single definition, there are many types of dependencies (Malone and Crowston 1994). However, those most often represented can be classified using the "single resource dependency hierarchy." Developed by Gilad Zlotkin, a post-doctoral fellow at CCS, this hierarchy appears in Figure 11 (Zlotkin 1994). The most general dependency, PRODUCERS-CONSUMERS-DEPENDENCY, is at the top, where a resource can have any number of producing activities and consuming activities. In the more specialized dependency types, the "Producer" and "Consumer" slots are redefined to enforce stronger constraints on the number of values allowed.

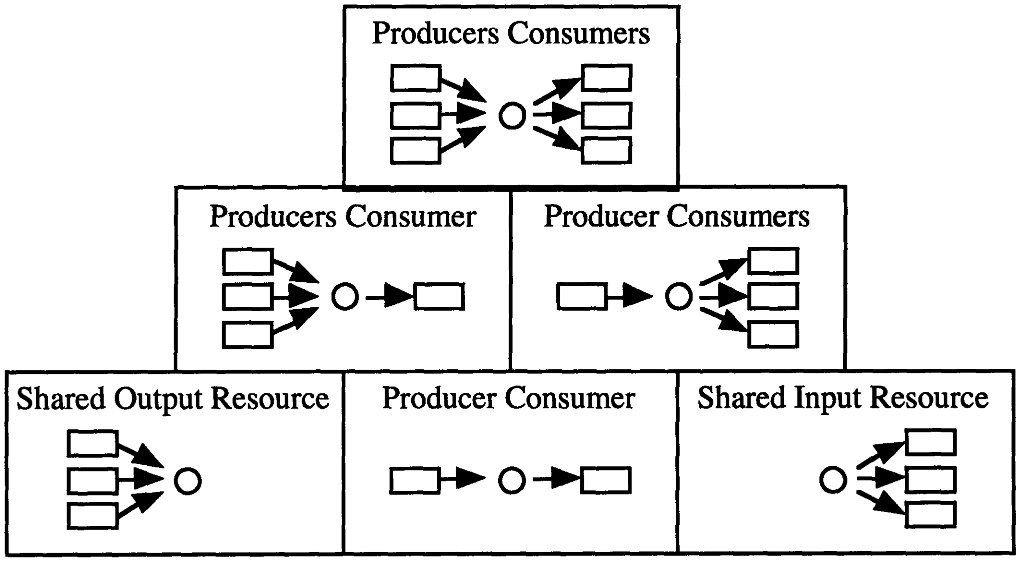

Figure 11: The single resource dependency hierarchy (Zlotkin 1994).

After determining which handbook entities require PIF class extensions, they must be described in frame syntax to be loaded by the translators. Not formally a kind of activity, the BUNDLE class is only a subtype of ENTITY. Currently, PIF does not have a

[20]Some of the translators for handbook prototypes are still in development.

way to represent a slot which can have values of either of two types. Therefore, though a bundle "Belongs-to" and "Contains" only activities and other bundles in theory, the strongest type restriction possible for these fields in PIF syntax is the ENTITY class. Though the stronger type check is not performed automatically by the toolkit, it can be done through other toolkit functions. One unconventional possibility would be to create a new abstraction ACTIVITY-OR-BUNDLE which has the subclasses ACTIVITY and BUNDLE. The values of the "Belongs-to" and "Contains" slots would then be constrained to this new type. However, type extensions are usually leaves on the class hierarchy.

The hierarchy of dependencies also brought some challenges in their representation. More specifically, it was not immediately clear which core type should be the superclass for these six new classes. Some people think that a producer/consumer dependency is just "syntactic sugar" for and could be defined as a set of simpler relations. In other words, for all producers P_i, consumers C_j, and resource R, a PRODUCERS-CONSUMERS-DEPENDENCY(P_i, C_j, R) object is composed of a CREATES(P_i, R) relation for each producer, a USES(C_j, R) relation for each consumer, and a PREREQUISITE(P_i, C_j) relation for each consumer.[21] (Each PREREQUISITE relation can contain all producers but only one consumer.) Others feel that this dependency type still embodies more, such as a coordination activity or mechanism. In either case, while one can definitely argue that dependencies are a useful abstraction, we would like to translate these types to some set of relations in systems that do not share the handbook extensions.

Unfortunately, there is currently no way to model this complex mapping in PIF frame syntax without using some of the more sophisticated language found in Ontolingua or KIF. While it is possible to say that any instance is composed of other instances (through the ENTITY "Component" slot), one cannot define the dependency class as being composed of other classes. Even if the type with one producer and one consumer, PRODUCER-CONSUMER-DEPENDENCY, could multiply inherit from all three CREATES, USES, and PREREQUISITE relations, our current class syntax does not allow us to map slots of a subtype to slots of supertypes. Therefore, because of these representation difficulties and because the composition of dependencies is an open question in our research group, PRODUCERS-CONSUMERS-DEPENDENCY is currently only a subtype of the more generic RELATION class.

[21]In a particular system, we might be able to infer the PREREQUISITE relation from the presence of both CREATES and USES relations.

5.2.2 Representing Process Specialization

After defining the handbook ontology, the most important decision was determining if process specialization should be represented as class specialization in PIF. Again, this is an open research issue. One school of thought believes that activities in the handbook can be treated just like class frames. If so, the handbook's specialization hierarchy would appear as a subtree of types originating from the core ACTIVITY type. Other systems can then use the handbook's representation of processes at different levels of abstraction. Also, handbook users can take advantage of other systems that have object-oriented inheritance by importing these classes.

This approach of representing handbook activities as classes is not without its disadvantages, however. First, in order for PIF users to make the most out of handbook data, they will need to share this large set of type extensions. In essence, sharing handbook data will not involve the exchange of any objects, and this runs counter to the intended use of PIF. Second, because simulations can only occur on actual instances of processes, these classes will have to be instantiated for any such operations. When exporting the instances for even a simple handbook decomposition, either the classes must also be shared or the "define-hierarchy" structure may have to contain a large part of specialization hierarchy.

Third, because bundles are not subtypes of the activities to which they belong, bundle layers must be removed from the specialization hierarchy. Bundles can still exist, but they must be transparent or supplemental to the activity class definitions. In other words, "Sell books" must be a subclass of and directly connected to "Sell Product" and not "How sold?" (in Figure 2). Finally, because PIF currently does not have multiple inheritance, activities that are specializations of more than one parent cannot be represented accurately.

The other school of thought maintains that process specialization is less rigid and more loosely defined than frame specialization and, therefore, should not be treated as such. Because of this and the difficulties mentioned above, we represent handbook activities as instances. In the future, with developments in the handbook and PIF, especially a more formal definition of specialization in the handbook (Wyner and Lee 1995), this decision may change. For now, handbook translators express this relationship through a "Specialization-of" attribute in ACTIVITY objects.

By translating the crux of the handbook data as ACTIVITY instances, our translators operate in stronger accordance with the intended usage of PIF. Until the specialization hierarchy is more established, the corresponding set of type extensions would be subject to much more change than those for bundles and dependencies. It would be inconvenient to load a frequently developing set of activity classes. While translators for a particular system will process a variety of instances, they are generally tuned for a relatively fixed set of class extensions. Additionally, because the "Specialization-of" field is specific to the handbook, it can also contain bundles, reflecting the specialization hierarchy in our prototypes more strongly. In summary, by sacrificing the conceptual elegance of equating both forms of specialization, PIF translations of handbook data are much simpler by not dealing with new classes.

6 Conclusion

Since the previous PIF translators made little attempt to solve the round trip problem, there was much room for improvement. The metric of any solution is the degree with which system-specific data is preserved across translations. With cooperating users and systems that meet the requirements of section 3.3, our framework allows unmodified data specific to a particular system to return in the same form. Additional work is necessary by the translator writer, but the ideal translator would completely reconstruct the system's representation based on the returned objects and attributes. If its state is not perfectly restored, however, we still have presented a mechanism whereby a user can examine the data and complete the translation manually if necessary. The key is that the data is still available to the importing translator.

Besides having a solution to the round trip problem, this work provides other benefits to the handbook research effort. Currently, PIF translators are the primary method of communication for handbook tools operating on different databases or platforms. They have been used to preserve the data from previous prototypes, and this allows our research group to continue to innovate and develop new tools. So long as each tool can import and export PIF files, its data will not be obsolete or inaccessible. This also allows handbook users to choose the tool that best suits their needs. Also, because the tools have access to data from others, we can evaluate them solely on their functionality.

Looking beyond the handbook, a toolkit which addresses this problem will promote the greater use of the PIF and, as a result, more exchange of business process descriptions. While one-way exchanges are useful for sharing data, this work makes it easier to actually make use another system and bring the results back. Even without a round trip solution, the toolkit is still useful. By making the ontology another input of the translators, we hope this encourages different parties to work together on developing type extensions. With the round trip solution, the translation process will be more complex not only for the computer but also for the user. Translators built with the toolkit should standardize the translator interface, making it as painless as possible. This may include the task of keeping track of which companion files belong to which system files.

Though the scope of this thesis was to present a solution for the handbook, we believe that the underlying principles of such a solution can be applied to other interchange formats and data-sharing scenarios. While the loss of attributes and type are specific to

PIF, the techniques of preserving data can certainly be used in other formats. For those formats that do not have a capability for system-specific data like PIF's "User-Attributes" field, a companion file can still be left on export instead of import. The round trip problem can still be averted, but the companion file will have to be shared with the interchange file. In situations where there is no interchange format, pairwise translators for two systems can still make use of companion files. In general, the companion concept is most useful when meaning preservation is of the highest priority. What we have is a paradigm where information that is not of immediate use is saved instead of discarded.

References

Ahmed, Erfanuddin. (1995). A Data Abstraction with Inheritance in the Process Handbook. Thesis for the degrees of Bachelor of Science in Computer Science and Engineering and Master of Engineering in Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Booch, Grady. (1994). *Object-Oriented Analysis and Design with Applications.* Second Edition. Benjamin/Cummings.

Genesereth, Michael R. and Richard E. Fikes (Editors). (1992). Knowledge Interchange Format Version 3.0 Reference Manual. Technical Report Logic-92-1, Computer Science Department, Stanford University.

Goh, Cheng Hian, Stuart Madnick, and Michael Siegel. (1994). Context Interchange: Overcoming the Challenges of Large-Scale Interoperable Database Systems in a Dynamic Environment. In *Proceedings of the Third International Conference on Information and Knowledge Management*, Gaithersburg, MD.

Gruber, Thomas R. (1993). A Translation Approach to Portable Ontology Specifications. *Knowledge Acquisition*, 5(2), 199-220. Ontolingua documentation available at ftp://ksl.stanford.edu/pub/knowledge-sharing/ontolingua.

Kernighan, Brian K. and Dennis M. Ritchie. (1988). *The C Programming Language.* Second Edition. Prentice-Hall.

Lee, Jintae and Thomas W. Malone. (1990). Partially Shared Views: A Scheme for Communicating among Groups that Use Different Type Hierarchies. *ACM Transactions on Information Systems*, 8(1), 1-26.

Lee, Jintae, Gregg Yost, et al. (1994). The PIF Process Interchange Format and Framework. CCS Working Paper #180, Center for Coordination Science, Massachusetts Institute of Technology.

Malone, Thomas W., Kevin Crowston, Jintae Lee, and Brian Pentland. (1993). Tools for inventing organizations: Toward a handbook of organizational processes. In *Proceedings of the 2nd IEEE Workshop on Enabling Technologies Infrastructure for Collaborative Enterprises*, Morgantown, WV.

Malone, Thomas W. and Kevin Crowston. (1994). The Interdisciplinary Study of Coordination. *ACM Computer Surveys*, 26(1).

Steele, Guy L., Jr. (1990). *Common Lisp: The language.* Second Edition. Digital Press.

Stroustrup, Bjarne. (1991). *The C++ Programming Language.* Second Edition. Addison-Wesley.

Tse, T. H. (1991). *A Unifying Framework for Structured Analysis and Design Models: An Approach using Initial Algebra Semantics and Category Theory.* Cambridge University Press.

Wyner, George M. and Jintae Lee. (1995). Applying Specialization to Process Models. Work in progress.

Zlotkin, Gilad. (1994). Coordinating Resource Based Dependencies. Process Handbook Research Seminar, Center for Coordination Science, Massachusetts Institute of Technology. October 14, 1994.

Appendix

Grammar for PIF files

```
pif_file:               <hierarchy> <instance_frame>*

hierarchy:              ( define-hierarchy <hierarchy_level>+ )
```

The ability to have multiple hierarchy roots is unique to this toolkit.
In strict PIF, there is only one hierarchy root (<hierarchy_level>).

```
hierarchy_level:        <class_id>
                        | ( <class_id> <hierarchy_level>* )
```

The asterisk (*) of hierarchy_level is a correction from (Lee and Yost 1994).

```
instance_frame:         ( define-frame <instance_id>
                                :own-slots ( <own_slot>* )
                        )

own_slot:               ( <slot_name> <slot_value>+ )

slot_value:             <constant>
                        | ( setof <constant>* )
                        | ( listof <constant>* )
                        | ( quote <s_expr> )
                        | '<s_expr>

constant:               STRING
                        | SYMBOL
                        | NUMBER

s_expr:                 <constant>
                        | (<s_expr>*)
                        | '<s_expr>
```

The RESTRICTED-KIF-SENTENCE is a limited subset of symbolic expressions.
See (Lee and Yost 1994) for details.

```
class_id:               SYMBOL
instance_id:            SYMBOL
slot_name:              SYMBOL
```

Grammar for ontology files

```
ontology_file:           <class_frame>*

class_frame:             ( define-frame <class_id>
                                 :own-slots ( <own_slot>* )
                                 :template-slots ( <template_slot>* )
                         )

template_slot:           ( <slot_name> <facet_or_slot_value>+ )

facet_or_slot_value:     (<facet> <slot_value>*)
                         | <slot-value>
```

Default slot values (i.e., <slot-value>) are not supported in this toolkit.

```
facet:                   SYMBOL
```

See section 4.2.2 for the facets supported in this toolkit.

PIF 1.0 in Frame Syntax

```
; The core PIF v1.0 ontology
; with corrections to slot tags that were also names of entities
; and aliases for these bad tags

(define-frame ACTIVITY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ENTITY)
            (documentation    "Many preconditions and postconditions can
be expressed in PIF without using the Precondition and Postcondition
attributes of ACTIVITY.  For example, the USE relation between activity
A and resource R implies that one of A's preconditions is that R is
available.  In general, the Precondition and Postcondition attributes of
ACTIVITY should only be used to express conditions that cannot be
expressed any other way in PIF.  Doing so will maximize the degree to
which a process description can be shared with other people.")
      )
      :template-slots (
            (Component        (slot-value-type ACTIVITY))
            (Precondition     (slot-value-type RESTRICTED-KIF-SENTENCE)
                              (maximum-slot-cardinality 1))
            (Postcondition    (slot-value-type RESTRICTED-KIF-SENTENCE)
                              (maximum-slot-cardinality 1))
            (Status           (slot-value-type SYMBOL))
      )
)

(define-frame ACTOR
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RESOURCE)
            (documentation    "An ACTOR represents a person, group, or
other entity (such as a computer program) that participates in a
process.")
      )
      :template-slots (
            (Has-Skill  (slot-value-type SKILL)
                        (slot-alias Skill))
      )
)

(define-frame CANNOT-BE-CONCURRENT
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "Activities among which this relation
holds cannot be executed concurrently.  Conversely, activities without
this relation among them are assumed to be concurrently executable.")
      )
      :template-slots (
            (Has-Activity     (slot-value-type ACTIVITY)
                              (slot-alias Activity))
      )
)
```

```
(define-frame CREATES
      :own-slots (
            (instance-of        CLASS)
            (subclass-of        RELATION)
            (documentation      "Activity CREATES resources.")
      )
      :template-slots (
            (Has-Activity       (slot-value-type ACTIVITY)
                                (maximum-slot-cardinality 1)
                                (slot-alias Activity))
            (Has-Resource       (slot-value-type RESOURCE)
                                (slot-alias Resource))
      )
)

(define-frame DECISION
      :own-slots (
            (instance-of        CLASS)
            (subclass-of        ACTIVITY)
            (documentation      "A DECISION is a special kind of activity
that represents conditional branching.  If the RESTRICTED-KIF-SENTENCE
in its If attribute is satisfied, the next activities in the process
flow are those specified in the SUCCESSOR relations in the Then
attribute.  If not, the next activities are those specified in the
SUCCESSOR relations in the Else attribute.")
      )
      :template-slots (
            (If          (slot-value-type RESTRICTED-KIF-SENTENCE)
                         (maximum-slot-cardinality 1))
            (Then        (slot-value-type SUCCESSOR))
            (Else        (slot-value-type SUCCESSOR))
      )
)

(define-frame ENTITY
      :own-slots (
            (instance-of        CLASS)
            (documentation      "ENTITY is the root of the PIF class
hierarchy.")
      )
      :template-slots (
            (Component          (slot-value-type ENTITY))
            (Documentation      (slot-value-type STRING)
                                (maximum-slot-cardinality 1))
            (Name               (slot-value-type SYMBOL)
                                (maximum-slot-cardinality 1))
            (User-Attribute     (slot-value-type SYMBOLIC-EXPRESSION))
      )
)
```

```
(define-frame GROUP
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ACTOR)
            (documentation    "A GROUP is a group of actors.  In PIF, a
GROUP is defined to be a special kind of ACTOR.  So, for example, one
can specify that some activity is performed by a group of people rather
than a single individual.  In the future, PIF will probably support
multiple inheritance.  When it does, GROUP will also be a subclass of a
SET type that supports general features of sets of objects.  For now, we
define GROUP as a subclass of ACTOR that defines a few additional set-
related attributes such as Cardinality and Member-Type.")
      )
      :template-slots (
            (Cardinality      (slot-value-type INTEGER)
                              (maximum-slot-cardinality 1))
            (Member           (slot-value-type ACTOR))
            (Member-Type      (slot-value-type CLASS))
      )
)

(define-frame MODIFIES
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "Activity MODIFIES resources.")
      )
      :template-slots (
            (Has-Activity     (slot-value-type ACTIVITY)
                              (maximum-slot-cardinality 1)
                              (slot-alias Activity))
            (Has-Resource     (slot-value-type RESOURCE)
                              (slot-alias Resource))
      )
)

(define-frame PERFORMS
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "Actor PERFORMS Activities.")
      )
      :template-slots (
            (Has-Actor        (slot-value-type ACTOR)
                              (maximum-slot-cardinality 1)
                              (slot-alias Actor))
            (Has-Activity     (slot-value-type ACTIVITY)
                              (slot-alias Activity))
      )
)
```

```
(define-frame PREREQUISITE
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "This relation represents that the
activities in Required-Activity must be completed before the activity in
Has-Activity can begin.  The distinction between the PREREQUISITE
relation and the SUCCESSOR relation is subtle.  PREREQUISITE represents
which activities must have been completed before another activity
begins, but does not specify precisely when those activities actually
occur.  SUCCESSOR represents the flow of control in a process -- that
is, it represents the precise order in which activities will actually
occur.")
      )
      :template-slots (
            (Required-Activity      (slot-value-type ACTIVITY))
            (Has-Activity           (slot-value-type ACTIVITY)
                                    (maximum-slot-cardinality 1)
                                    (slot-alias Activity))
      )
)

(define-frame RELATION
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ENTITY)
            (documentation    "Currently, RELATION objects have no
attributes of their own.  PIF uses it as an abstract parent class for
more specific relation classes such as USES and PERFORMS.")
      )
)

(define-frame RESOURCE
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ENTITY)
            (documentation    "A RESOURCE is an entity that is needed in
some way to perform an activity.  This includes people (represented by
the ACTOR subclass in PIF), physical materials, time, and so forth.  The
PIF Working Group has discussed adding attributes such as Consumable,
Sharable, and so forth, but so far no decision has been made on what
attributes are appropriate.")
      )
)

(define-frame SKILL
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RESOURCE)
            (documentation    "A SKILL object represents expertise
possessed by actors.  As yet, we have not decided on any attributes that
are unique to SKILL objects, as the examples we have used in developing
PIF make little use of skill information.")
      )
)
```

```
(define-frame SUCCESSOR
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "An activity X is a SUCCESSOR of activity
Y if X begins after Y completes and no other activities occur between X
and Y.  A successor relation is unconditional (that is, activity X
always immediately follows activity Y) unless it appears as a value of
either the Then or Else attribute of a DECISION object.  Intuitively,
SUCCESSOR relations corresponds to the flow arrows in a process
flowchart.  The SUCCESSOR and PREREQUISITE relations are subtly
different.  See the description of PREREQUISITE for details.")
      )
      :template-slots (
            (Has-Predecessor  (slot-value-type ACTIVITY)
                              (maximum-slot-cardinality 1)
                              (slot-alias Predecessor))
            (Has-Successor    (slot-value-type ACTIVITY)
                              (slot-alias Successor))
      )
)

(define-frame USES
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
            (documentation    "Activity USES resources.")
      )
      :template-slots (
            (Has-Activity     (slot-value-type ACTIVITY)
                              (maximum-slot-cardinality 1)
                              (slot-alias Activity))
            (Has-Resource     (slot-value-type RESOURCE)
                              (slot-alias Resource))
      )
)
```

Process Handbook extensions to PIF

```
; See section 5.2.1 for details.

(define-frame BUNDLE
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      ENTITY)
      )
      :template-slots (
            (Belongs-to       (slot-value-type ENTITY))
            (Contains         (slot-value-type ENTITY))
      )
)

; Coordinating Resource Based Dependencies
;
; Gilad Zlotkin
;
; Process Handbook Research Seminar
; October 14, 1994

(define-frame PRODUCERS-CONSUMERS-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      RELATION)
      )
      :template-slots (
            (Producer         (slot-value-type ACTIVITY))
            (Consumer         (slot-value-type ACTIVITY))
            (Has-Resource     (slot-value-type RESOURCE)
                              (maximum-slot-cardinality 1))
      )
)

(define-frame PRODUCERS-CONSUMER-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      PRODUCERS-CONSUMERS-DEPENDENCY)
      )
      :template-slots (
            (Consumer   (slot-value-type ACTIVITY)
                        (maximum-slot-cardinality 1))
      )
)
```

```
(define-frame SHARED-OUTPUT-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      PRODUCERS-CONSUMER-DEPENDENCY)
            (documentation    "This dependency is also known as the
common object dependency.")
      )
      :template-slots (
            (Consumer   (slot-value-type ACTIVITY)
                        (slot-cardinality 0))
      )
)

(define-frame PRODUCER-CONSUMERS-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      PRODUCERS-CONSUMERS-DEPENDENCY)
      )
      :template-slots (
            (Producer   (slot-value-type ACTIVITY)
                        (maximum-slot-cardinality 1))
      )
)

(define-frame SHARED-INPUT-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      PRODUCER-CONSUMERS-DEPENDENCY)
      )
      :template-slots (
            (Producer   (slot-value-type ACTIVITY)
                        (slot-cardinality 0))
      )
)

(define-frame PRODUCER-CONSUMER-DEPENDENCY
      :own-slots (
            (instance-of      CLASS)
            (subclass-of      PRODUCERS-CONSUMERS-DEPENDENCY)
            (documentation    "When PIF has multiple inheritance, this
class should really be a subclass of both PRODUCERS-CONSUMER-DEPENDENCY
and PRODUCER-CONSUMERS-DEPENDENCY classes.")
      )
      :template-slots (
            (Producer   (slot-value-type ACTIVITY)
                        (maximum-slot-cardinality 1))
            (Consumer   (slot-value-type ACTIVITY)
                        (maximum-slot-cardinality 1))
      )
)
```