View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by NASA Technical Reports Server

N94-11430

Reuse — A Knowledge-Based Approach

Neil Iscoe, Zheng-Yang Liu, Guohui Feng
EDS Research, Austin Laboratory
1601 Rio Grande, Ste. 500
Austin, Texas 78701
(512) 477-1892
iscoe@austin.eds.com

© Copyright 1992, EDS

Introduction

Although empirical field studies [Curtis 88] have shown that application domain knowledge is critical to the success of large projects, this knowledge is rarely stored in a form which facilitates its use in creating, maintaining and evolving software systems. The reason for this is that application domain knowledge is implicitly embodied in application code rather than explicitly recorded and maintained in separate documents. Even when documents are maintained separately from the code, the knowledge is stored in voluminous natural language documents in an informal rather than a formal manner. Although problem-specific languages, sometimes called 4th generation or nth generation languages, are designed to remedy this situation, these languages capture domain-specific knowledge in an ad hoc manner which usually does not allow the results to be generalized or even replicated. Capturing and managing this knowledge is a prerequisite to reusing components in software development.

Consider EDS, a company which produces large software systems for a variety of industries such as utilities, finance, or health insurance. Associated with each industry area is a body of knowledge which is critical to specifying and implementing software systems. This knowledge includes legal, financial, technical, and other expertise which is acquired by personnel over a period of many years. EDS is organized into strategic business units (SBUs) so that the business unit's knowledge about a particular industry can be leveraged through reuse of application knowledge. Although organizational management is an efficient and effective method of gaining productivity, additional gains can be realized by automating portions of the software production cycle.

While system engineers make extensive use of several existing CASE tools, these tools help streamline but do not substantially change the mapping process from concept to implementation. The problem is that today's methodologies, languages, and CASE tools do not take advantage of the domain-specific knowledge associated with particular industries. This is because this knowledge is rarely available in a form in which it can be reused.

We are attempting to capture the domain-specific knowledge about different industry areas as a set of application domain models. Application domain models are representations of relevant aspects of application domains that can be used to achieve specific software engineering operational goals such as elicitation of specifications, code generation, and reverse engineering. Operational goals are always implicit in the construction of a domain model and are essential to understanding the form and content of that model. Unlike generalized knowledge representation projects such as Cyc [Lenat 90] that attempt to provide a basis for modeling encyclopedic knowledge, domain modeling explicitly acknowledges the commonly held view [Amarel 68] that representations are designed for particular purposes. These purposes—the operational goals—inherently bias any particular solution and dictate the final form of the model.

Many different operational goals and modeling projects are being pursued within the field of domain modeling [Iscoe 91]. A model is the result of conscious decisions about what to describe and what to ignore. No model is complete or correct in the sense that it is applicable to all tasks. Domain models in our system are structured to represent the type of information that is used within EDS SBUs to achieve our operational goals. Although EDS serves a wide range of industries, we are not attempting to model real-time or other application areas which diverge from standard business transaction processing. However, we are modeling all the aspects of these programs which include the rules, integrity constraints, type definitions, and policy decisions.

Figure 1 illustrates our view of the process. First, in order to build a domain model, we must acquire domain-specific knowledge from various sources and synthesize it into a domain model. At this stage, we focus on digesting, understanding, and formalizing the gathered information. Second, in order to build specification models for various applications, we must select relevant pieces of domain knowledge from the domain model. In this phase, we concentrate on tailoring general domain knowledge to the specific application at hand.

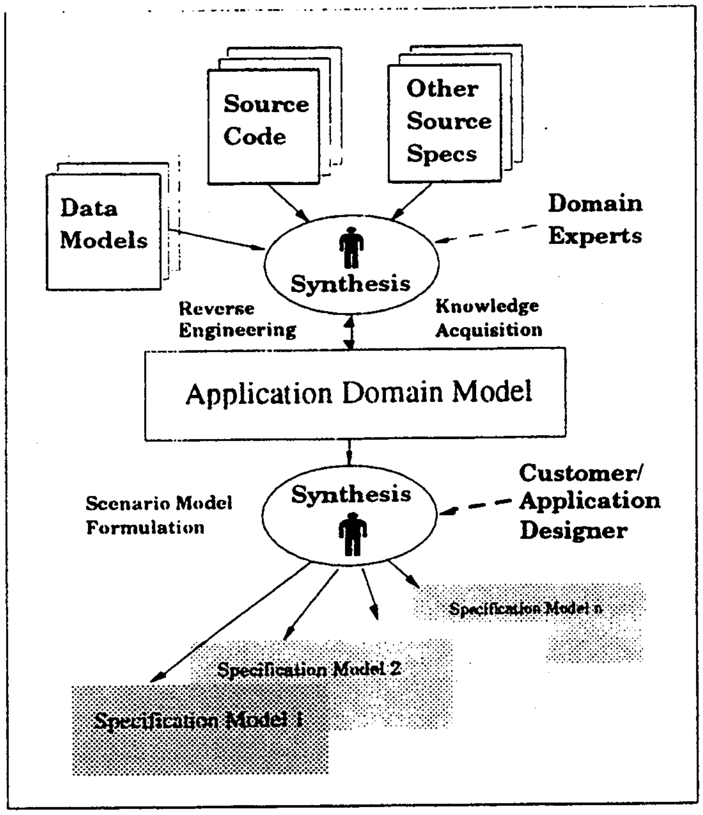


Figure 1 — Software Synthesis

Our approach to gathering domain knowledge is based on synthesizing information through a combination of manual, semi-automated, and automated techniques. Manually acquiring application knowledge by interviewing domain experts is a laborious, costly, and error-prone process. Consequently, we attempt to reverse engineer all relevant existing application artifacts and use experts only when necessary to make sense out of and resolve conflicting information, to supply missing information, and to add new knowledge to the model.

The reality of program development is that relatively few programs are ever conceived entirely from scratch. Instead, they are created within the context of existing manual and automated systems. New application programs are created because technological, regulatory, economic, and other business and societal reasons make existing programs obsolete. Although old programs are generally outdated and archaic they still provide a potential source of information. The portions of the knowledge acquisition process which we are automating are driven by the availability of existing source code, data models, and other machine-readable sources of specifications such as data dictionaries.

In Figure 1, the double-headed arrow between the top synthesis oval and the application domain model rectangle indicates that the application domain model is also used as a source of information in the synthesis process. As in any bootstrapping process, the more a system knows, the easier it is to load with additional information. At this stage of the synthesis process, human interaction is supplied by application or "subject matter" experts who have specialized knowledge of an application domain gained through exposure to multitudes of programs within an application area. A Truth Maintenance System and Theorem Prover ensures domain model consistency and generates queries to the designer concerning potential inconsistencies. The final result of the first stage of the software synthesis process is an application domain model.

The next stage of synthesis is the production of actual application specifications. Although the programs within a specific application domain share the same general legal, physical, and economic constraints, the construction of any particular program specification depends upon a localized set of specification decisions driven by the enduser or customer. Furthermore, requirements and their resultant specifications change. Even when specifications are correctly gathered and refined, the nature of most real world applications is that requirements and specifications evolve throughout the existence of a program. These modifications are neither caused by the sometimes mercurial nature of endusers nor by sloppiness in the original design process but are the result of natural occurrences such as workload modifications, environmental changes, new technologies, economic disturbances, legal mandates, and so on, that remove old requirements and create consequent new specifications.

The bottom portion of Figure 1 illustrates that multiple *specification models* can be created within an application domain. These "models" or application program specifications are consistent and correct with respect to the application domain model although they themselves may not be consistent with each other.

Implementation

Figure 2 is a schematic view of the knowledge acquisition synthesis system which is implemented using X windows on a SPARC II client/server architecture. Domain experts interact with the system to store their knowledge into a domain model. The arrowheads attached to the languages at the bottom portion of the figure show our interactions with particular specification languages. Domain and specification model consistency is maintained by a specialized theorem prover. The theorem prover, *STR+VE*, is an upgraded version of the prover presented in [Bledsoe 80] for proofs of theorems in general inequalities. A specialized inconsistency detection and correction system [Feng 93] is being constructed to interface between the modeling system and the theorem prover.

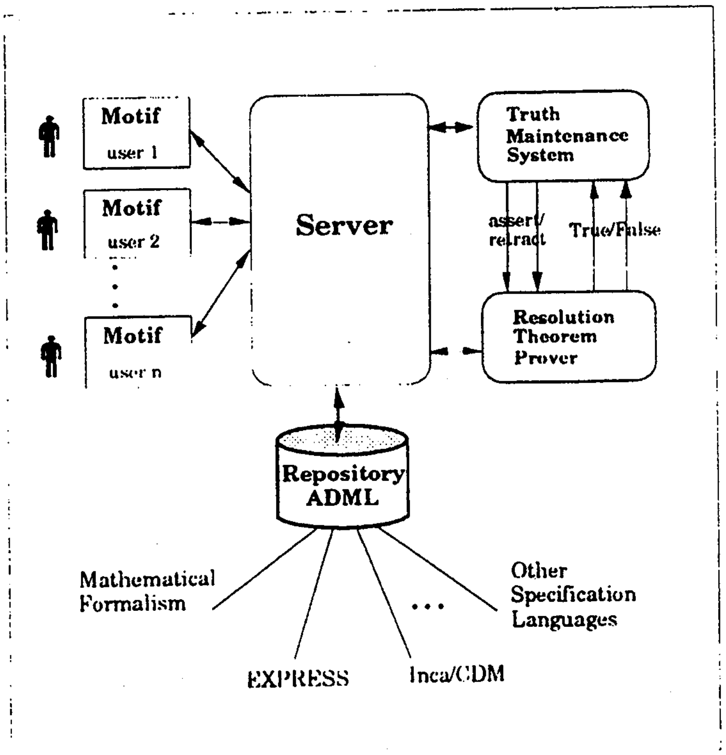


Figure 2 — Schematic Knowledge Acquisition System

Dynamic Knowledge Structure

Hierarchies are a natural way to view and organize information and, at some level of abstraction, are a part of most object-oriented and knowledge representation languages. Although hierarchical organizational strategies provide a reasonable way to structure knowledge within complex domains, the creation of a hierarchical structure, like any type of representational scheme, imposes a particular view of the world. Unfortunately, the simplicity of the concept can sometimes obscure the semantics that a model is attempting to capture. By carefully analyzing cases and building the appropriate subtypes, a variety of entity-relationship and object-oriented modeling and programming techniques can be used to capture information [Booch 91].

When the project is small and the view is simple, or when the world is static and the view doesn't change, a static hierarchy is a reasonable way to represent cases. But in large projects, there is no particular view that is optimal for every class of enduser or every application program. When all possible partitionings are made explicit, case transparency becomes impossible. The monolithic tree structure created by case expansion obscures relevant and interesting cases. Furthermore, program specifications change at a rapid enough pace that static hierarchies are not sufficient to capture the process of system evolution. A paradox of object-oriented approaches is that well adapted structures are not adaptable to new situations.

As an example, consider software systems that manage the payment of health insurance claims. Although conceptually simple, these systems—like all business transaction systems—must contain detailed processing information necessary to handle hundreds of thousands of different cases created by the appropriate partitioning of dozens of attributes such as gender, marital_status, age, previous_condition, employment, deductibles, copayments, prognosis, and so on.

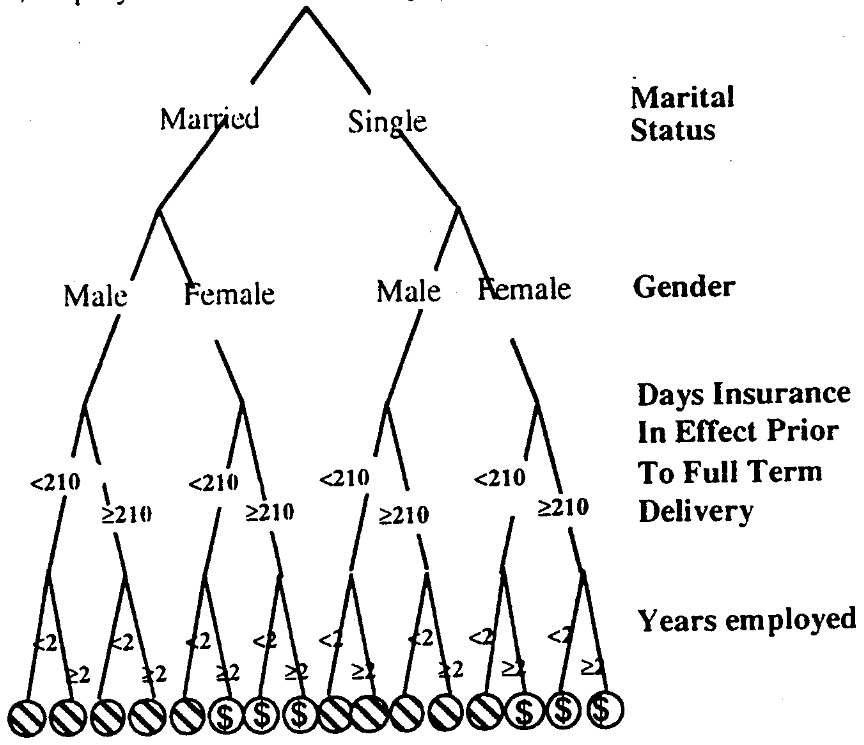


Figure 3 — Eligibility for Maternity Benefit $

Figure 3 is a hierarchy that represents only four attributes and their relevance to *Maternity_benefit$* in a company's health insurance policy. The circled leaf nodes represent cases. Dollar signs within circles represent benefit eligibility while the slash marks represent cases in which the employees are ineligible to receive maternity payments. In this particular case, marital status has no effect on eligibility. Gender is obvious enough to be ignored in the case of maternity benefits but is included in this and a variety of other health claim cases as an additional check against errors and fraudulent claims. Finally, *Days Insurance In Effect Prior To Full Term Delivery* (DIPFT) is used to

C-3.

create two groups; females with greater than or equal to 210 DIPFT receive benefits while those less than 210 DIPFT do not. However, if an employee has worked for the company for more than two years, the DIPFT restriction does not apply.

Figure 3 can be simplified to Figure 4 which eliminates the irrelevant attribute of marital_status and shows only the relevant decision nodes. In this uncomplicated example, the dollar eligibility for maternity benefits can be determined by the examination of only three attributes. Although Figure 4 is simple, it is easy to see how with only two dozen binary style attributes the complete specification hierarchy contains a combinatorial explosion of over ten million leaf nodes. This type of representation makes subtree identification and case analysis difficult and bug correction a painful case-by-case (node-by-node) process.

Term subsumption systems such as CLASSIC [Borgida 89] automate this process by determining the place in a hierarchy in which terms are subsumed. But subsumption systems assume a single structure in which all sub-models can belong. In the case of applications such as health insurance, individual program module specifications may have different hierarchical structures and still maintain the integrity and constraint rules of the domain model.

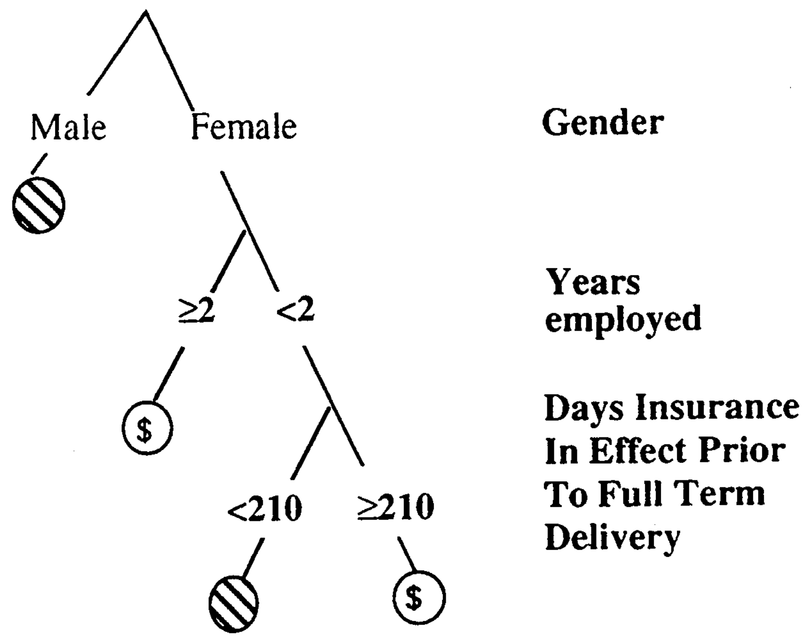


Figure 4 — Simplified Maternity Benefit Hierarchy

Our approach is to dynamically create the appropriate minimal hierarchical structure at the time that it is necessary to view a model or create a specification. This accomplishes two goals. It gives domain experts and program designers a more focused way of viewing application specifications, and it assures that the completed specifications will always contain the necessary attributes and value constraints.

Attributes

In order to automate the creation of minimal hierarchies such as the one shown in Figure 4, we begin with a careful consideration of the data elements or attributes that are used to distinguish cases from one another. For the purposes of this paper, assume that a class is a type definition which is used to instantiate objects (i.e. Person is a class; Jim and Suzy are objects). One way to view an attribute is as a function which defines how a set of objects is mapped within a class. As an example consider the attribute gender as a function which separates people into categories—males and females. This type of attribute is a *total* function over the class of people. That is, it applies without qualification to all people within the model.

Unlike total functions, *partial* functions do not apply to all objects within a class and have consequent restrictions placed on their usage. *Maternity_benefit$* is a *partial* function which only applies to females who have been employed for two years or who have greater than 210 DIPFT. The necessary constraints for the correct application of partial functions is one of the items captured by a domain model.

We capture information about total and partial functions in terms of conditional and unconditional attributes. As illustrated in Figure 1, domain models contain information which is used to create a variety of specification models. Although *total* functions apply to all objects of a class, not all attributes need appear in a particular class. Attributes which must appear in a class such as social_security_number for an employee, or weight and sales_price for an inventory item are differentiated from those which although they are independent of other attributes might appear in a class. A must_have attribute `Attrib` is an attribute that is required to occur in class `Entity_Class` for all specification models. It is represented formally as follows:

```
(must_have Entity_Class Attrib)
   →∀spec_model (used spec_model Entity_Class Attrib)
```

An applicable `Attrib` is an attribute that is not necessarily included in all specification models. However, it is included in the domain model because it has been observed in at least one specification model.

```
(applicable Entity_Class Attrib)
    →∃spec_model (used spec_model Entity_Class Attrib)
```

Conditional applicable attributes are *partial* functions such as *Maternity_benefit$*. In general they are defined in such a way that their selection by an application designer forces the use of the attributes which are required for their proper invocation. The formal definition of cond_applicable is as follows:

```
(cond_applicable Class Attr (P a val))
→∀spec_model, object
    ((used spec_model Class Attr)
    →
    (used spec_model Class a) ∧
    ((instance class object) ∧
     (<> (attr object) UNDEF)
     → (p (a object) val)))
```

Instantiating the definition with Person for Class, *Maternity_benefit$* for Attr, and *(& (= Gender Female)(OR (≥ DIPFT 210) (≥ Years_employed 2)))* for *(P a val)* yields:

```
(cond_applicable Person
   Maternity_benefit$
    (& (= Gender Female)
      (OR (≥ DIPFT 210)
           (≥ Years_employed 2))))
```

The implication then produces the following representation which contains the information which was used to produce the simplified hierarchy shown in Figure 4.

```
(→ (used spec_m Person Maternity_benefit$)
   ((used spec_m Person Marital_status) ∧
    (used spec_m Person Gender) ∧
(used spec_m Person Years_employed) ∧
((instance Person object) ∧
 (<> (Maternity_benefit$ object) UNDEF)
 →(& (= (Gender object) Female)
      (OR (≥ (DIPFT object) 210)
      (≥ (Years_employed object) 2))))·)
```

The preceding representation is also used directly by the automated theorem prover to refuse to allow inconsistent assertions to be made. For example, an assertion such as

```
(applicable Person Maternity_benefit$)
```

would not be allowed because *Maternity_benefit$* is a partial and not a total function.

Reverse Engineering Data Models

We use reverse engineering to build domain models by analyzing the data gathered from the applications of an entity-relationship-based CASE tool that is used by EDS SBUs for data modeling and code generation. By analyzing these data models, we have access to tens of thousands of specific descriptions of entities, relationships, and constraints which have been used to specify application programs. The knowledge acquisition process via reverse engineering is assisted with automated tools that identify the classes of objects, relationships, and events of the application domain. It translates existing data models with enhancements by producing dictionaries of data elements, classes, and constraints. Four types of enhancements are made:

- Restructuring—Attributes of a class grouped as must_have, applicable, and conditional_applicable attributes.
- Clustering—A set of data elements as a conceptual unit.
- Retyping—Entities and relations into classes. Data elements into scaled attributes.
- Input from Domain Expert—New features, measurement, constraints.

We are also attempting to reverse engineer source code to capture additional constraints. Our ongoing reverse engineering work has other subgoals besides instantiating domain models. While this project can currently produce data element and class definitions, it is not yet capable of producing more sophisticated domain modeling information.

Interactive Process

In our approach, the process of extracting specific application domain knowledge is interactive so that, at points, a domain expert can override the system's choice and provide new information. For example, each data element is assigned a set of constraints reflecting the domain requirements and is represented as a scale of measurement. A quantity data element is described with *min* and *max* values, *dimension*, *measurement unit* and *granularity*. When these pieces of information are not available in the original code, the system makes a guess by analyzing the textual description associated with the data element through a key-word recognition method. For example, when a data description contains any of the words "miles, distance, width," then the program guesses that the quantity is of dimension *length*. Sometimes, more than one guess is possible. In this case, the system prompts the user for input. The system remembers the user's choice and incrementally expands its key-word bank. If a similar situation occurs later on with another data element, the information would be used in determining the properties of the other data element. For each dimension, there is a set of corresponding measurement units, such as "meter, centimeter, foot, inch" and so on for *length*, from which the user is free to choose to assign to a quantity data element of the dimension.

Two advantages of this representation are apparent. First, the system ensures that the data are used in accordance with their definitions. For example, a careless mistake of adding a quantity of dimension *length* to another quantity of dimension *time* will be caught. Most CASE tools available today cannot catch these kinds of problems simply because they do not represent the knowledge. Second, we can expand the range of an application program by allowing a customer to choose his or her preferred choice of expression, as long as a data element does not violate its given constraints.

For example, given a *length* data element, one can choose either mile, or kilogram, or any other unit in the dimension as the measurement unit. The system can easily translate different units within a dimension.

Summary

This paper has described our research in automating the reuse process through the use of application domain models. Application domain models are explicit formal representations of the application knowledge necessary to understand, specify, and generate application programs. Furthermore, they provide a unified repository for the operational structure, rules, policies, and constraints of a specific application area.

In our approach, domain models are expressed in terms of a transaction-based meta-modeling language. This paper has described in detail the creation and maintenance of hierarchical structures. These structures are created through a process that includes reverse engineering of data models with supplementary enhancement from application experts. Source code is also reverse engineered but is not a major source of domain model instantiation at this time.

In the second phase of the software synthesis process, program specifications are interactively synthesized from an instantiated domain model. These specifications are currently integrated into a manual programming process but will eventually be used to derive executable code with mechanically assisted transformations.

This research is performed within the context of programming-in-the-large types of systems. Although our goals are ambitious, we are implementing the synthesis system in an incremental manner through which we can realize tangible results. The client/server architecture is capable of supporting 16 simultaneous X/Motif users and tens of thousands of attributes and classes. Domain models have been partially synthesized from five different application areas.

As additional domain models are synthesized and additional knowledge is gathered, we will inevitably add to and modify our representation. However, our current experience indicates that it will scale and expand to meet our modeling needs.

Acknowledgments

We wish to thank Betty Milstead, Srinivas Nedunuri, and Britt Yenne for their comments on earlier drafts of this paper.

References

Agresti, W. (1986). "What are the new paradigms?" in *New Paradigms for Software Development* (eds. Agresti, W.W.), IEEE Computer Society, Washington, D.C., 6-10.

Agresti, W. (1986). "Framework for a flexible development process" in *New Paradigms for Software Development* (eds. Agresti, W.W.), IEEE Computer Society, Washington, D.C., 11-14.

Amarel, S. (1968). "On representations of problems of reasoning about actions" in *Machine Intelligence*, American Elsevier, New York.

Barstow, D.R. (1984). "A display-oriented editor for INTERLISP" in *Interactive Programming Environments*, McGraw-Hill Book Company, New York.

Bledsoe, W.W. and Hines, L.M. (1980). "Variable elimination and chaining in the resolution-base prover for inequalities" in *Proceedings of the 5th Conference on Automated Deduction*, Springer-Verlag, Les Arcs, France, 70-87.

Boehm, B. (1976). "Software engineering" in *IEEE Transactions on Computers*, Vol. C-25, No. 12 (December 1976), 1226-1241.

Booch, G. (1991). *Object-Oriented Design with Applications*. Colorado Springs, CO: Benjamin/Cummings Publishing Company, 1991.

Borgida, A., Brachman, R.J., McGuinness, D.L., and Resnick, L.A. (1989). "CLASSIC: A structural data model for objects" in *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data*, 59-67.

Curtis, B., Krasner, H., and Iscoe, N. (1988). "A field study of the software design process for large systems" in *Communications of the ACM*, Vol. 31, No. 11, 1268-1287.

Davis, A.M., Bersoff, E.H., and Comer, E.R. (1988). "A strategy for comparing alternative software development life cycle models" in *IEEE*, Vol. 14, No. 10, 1453-1461.

Feng, G. (1993). "Maintaining the integrity of schema of knowledge-based management system by circumscription," submitted to Second Symposium on Logical Formalizations of Commonsense Reasoning, Austin, TX, January 11-13, 1993.

Iscoe, N., Williams, G.B., and Arango, G. (1991). "Domain modeling for software engineering" in *Proceedings of 13th International Conference on Software Engineering—Domain Modeling Workshop*, Austin, TX, 1-4.

Lenat, D.B., Guha, R.V., Pittman, K., Pratti, D., and Shepherd, M. (1990). Cyc: toward programs with common sense" in *CACM*, Vol. 33, No. 8, 30-49.

McCracken, D. and Jackson, M. (1981). "A minority dissenting position" in *Systems Analysis and Design—A Foundation for the 1980's* (eds. Cotterman, W.W., et al), Elsevier North-Holland, New York, 551-553.

Royce, W. (1970). "Managing the development of large software systems: concepts and techniques," in *Proceedings of WESCON*, August 1970.

Swartout, W.R. and Balzer, R. (1982). "On the inevitable intertwining of specification and implementation" in *CACM*, Vol. 25, No. 7, 438-446.

Reuse — A Knowledge-Based Approach
(Application Domain Modeling)

Neil Iscoe, Zheng-Yang Liu, Guohui Feng

EDS Research
Austin Laboratory for
Software Engineering & Computer Science

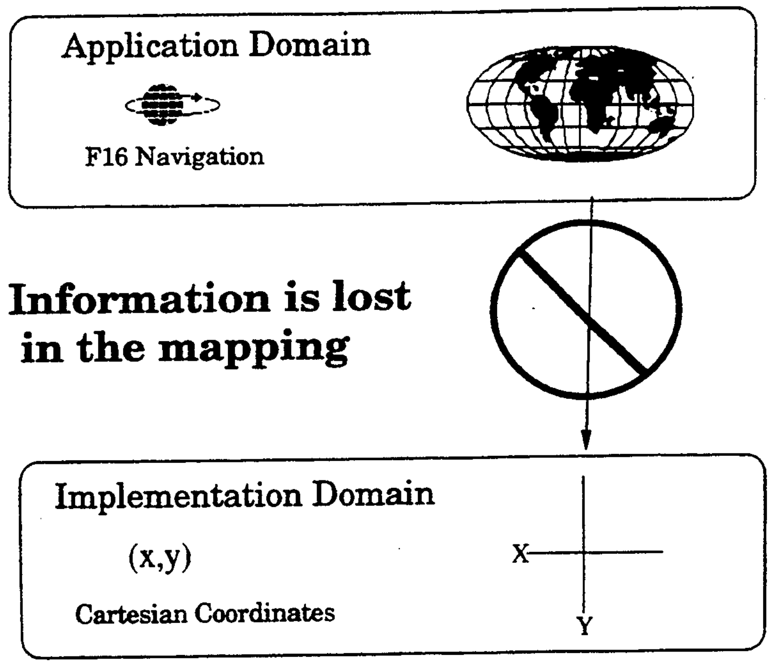
© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted

Reuse — A Knowledge-Based Approach
(Application Domain Modeling)

- What We Mean By Application Domain Knowledge
- What Application Domain Models Are
- Why We Care
- Why You Should Care
- What We're Doing

© Copyright 1992. EDS. All rights reserved. Permission to copy must be explicitly granted.

Why Do We Care About Capturing & Reusing Knowledge?



© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted

MCC Field Study
An empirical study of large software projects.

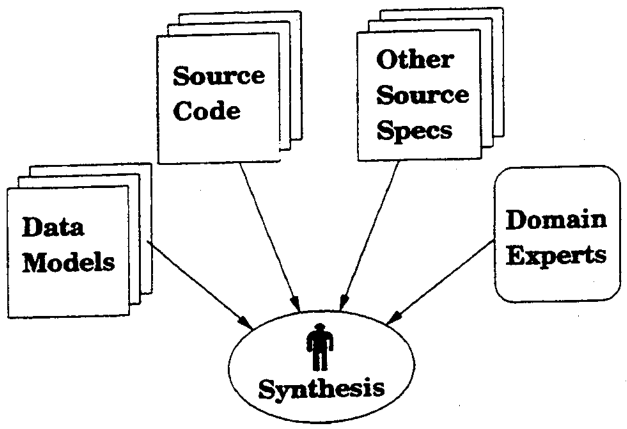
- 9 companies
- 17 projects
  Ranging in size from 24K to 10M LOC
- 97 interviews
- 3,500 pages of transcripts

Application Domain Knowledge

- was critical to the success (or failure) of the project
- was thinly spread throughout the organization
- was rarely written down in any concise form

© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted

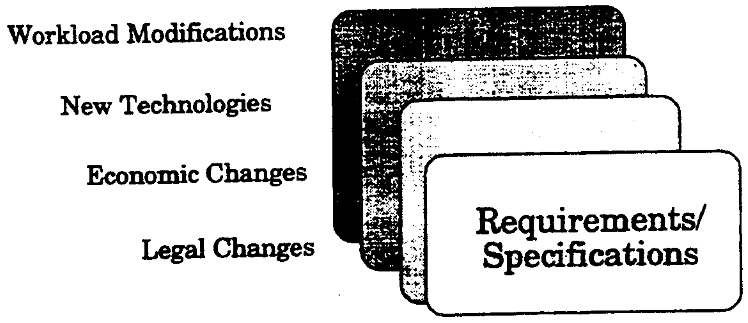
Application Knowledge is Rarely Concisely Stored



© Copyright 1992. EDS: All rights reserved: Permission to copy must be explicitly granted.

Requirements & Specifications Change Over Time
Static Specifications Don't Exist

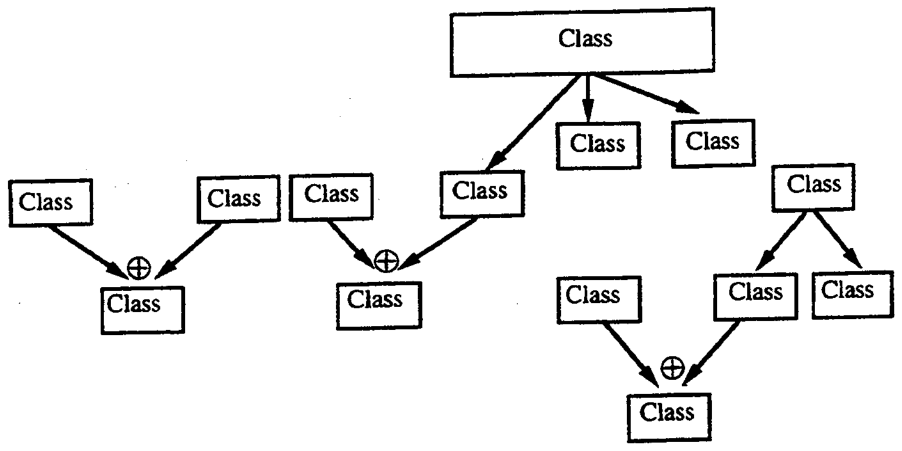
- Using A Program Changes One's View Of The Problem
- People Change Their Minds
- The World Changes



System Evolution Must Be Handled In A Disciplined Fashion

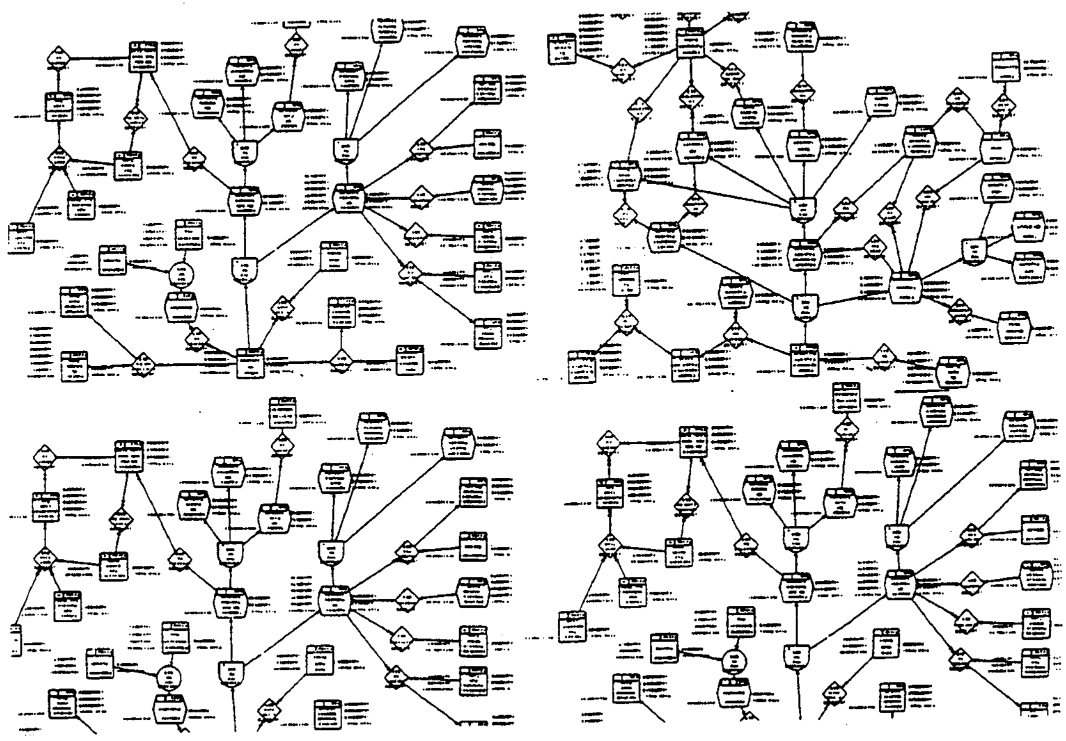
© Copyright 1992. EDS: All rights reserved: Permission to copy must be explicitly granted

Organization/Structure Is A Type Of Knowledge



© Copyright 1992 EDS. All rights reserved; Permission to copy must be explicitly granted.

Real Models Can Be Very Complicated



Why Do We Care About Capturing & Reusing Knowledge?

Application Knowledge is:

- Lost In The Mapping From Specification To Implementation
- Critical To The Success (Or Failure) Of Large Projects
- Thinly Spread Throughout Organizations
- Rarely Written Down In Any Concise Form

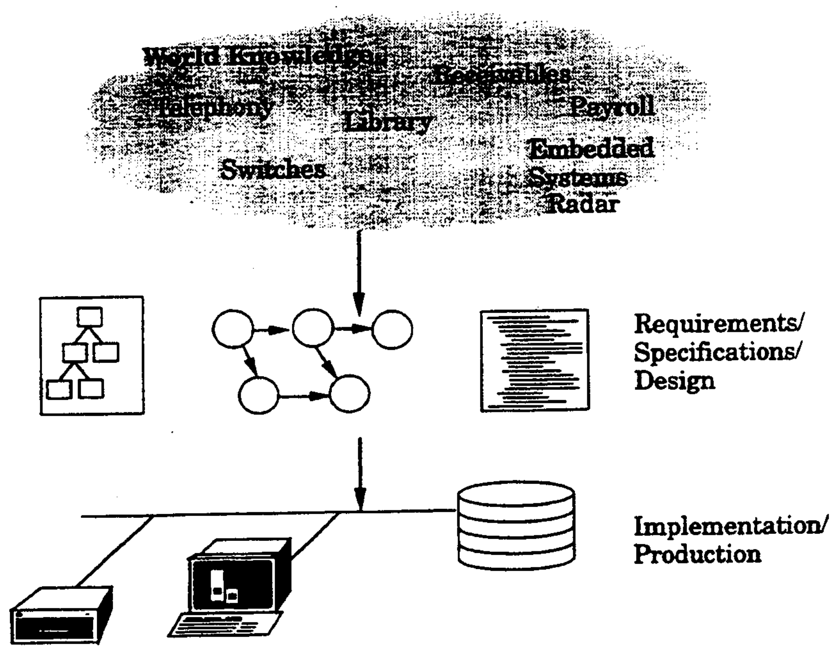
Requirements & Specifications Change

Organizational/Structural Knowledge Is Complex

Multiple Subsystems Require Multiple Views

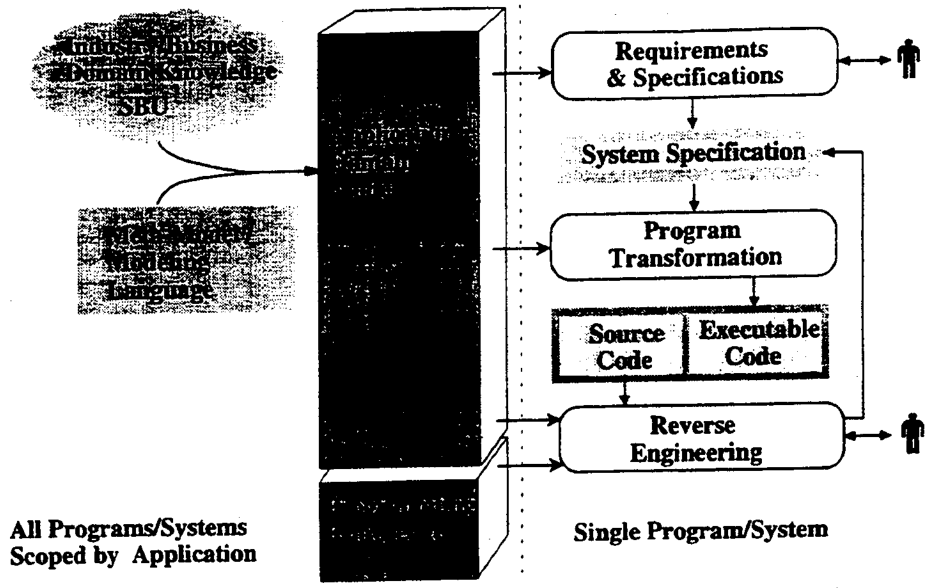
© Copyright 1992 EDS. All rights reserved: Permission to copy must be explicitly granted

Old Way: Single Problem - Single Solution



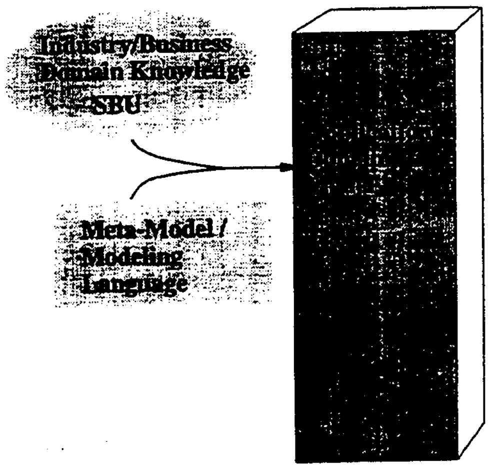
© Copyright 1992 EDS. All rights reserved. Permission to copy must be explicitly granted

New Way: Knowledge is Stored in A Model and Reused


Industry/Business Domain Knowledge SBU
Meta-Model/ Modeling Language
Requirements & Specifications
System Specification
Program Transformation
Source Code
Executable Code
Reverse Engineering
All Programs/Systems Scoped by Application
Single Program/System

© Copyright 1992. EDS. All rights reserved: Permission to copy must be explicitly granted

Application Knowledge Is Stored In A Domain Model


Industry/Business Domain Knowledge SBU
Meta-Model / Modeling Language

© Copyright 1992. EDS. All rights reserved: Permission to copy must be explicitly granted

How Do We Get Application Domain Knowledge Into Our Systems?

Old Way: Domain Knowledge is embedded in code.

- Hard to change the system
- Knowledge is hidden from view

New Way: Domain Knowledge is separated from the system.

- Can modify the system
- Knowledge is clearly separate

© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted

Application Domain Models Are Formal Representations Of Knowledge Designed To Achieve Specific Operational Goals:

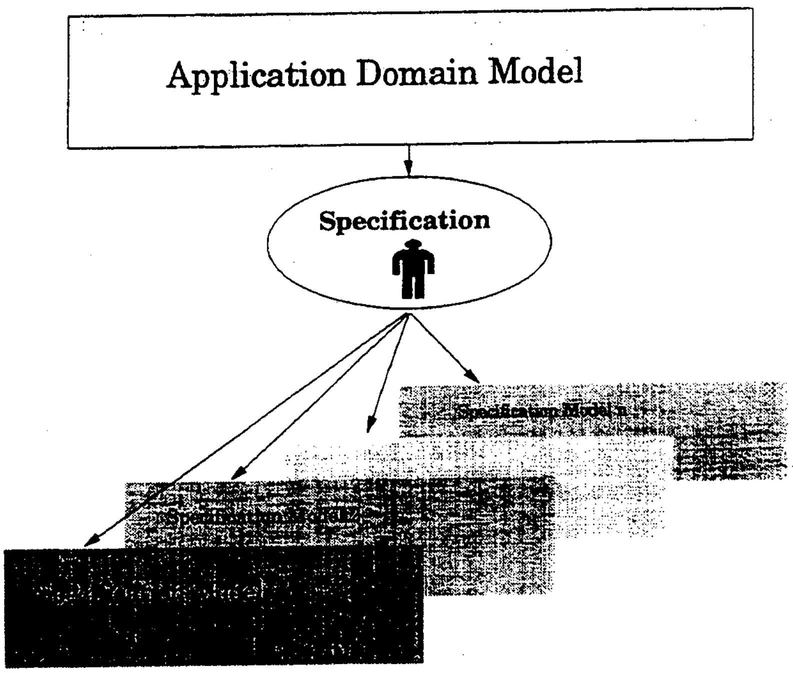
- **Requirements & Specifications**
  Eliciting, verifying, and formalizing software requirements and specifications

- **Automated Program Generation**
  Generating code from a system specification

- **Reverse Engineering**
  Identifying the semantics of existing code

© Copyright 1992. EDS: All rights reserved. Permission to copy must be explicitly granted

Many Specification Models Can Be Created From An Application Domain Model



Essential Characteristics

Application Domain Models & Specification Models:

- are formal structures,
- are computationally tractable,
- allow for reasoning and inference to support specific operational goals.

© Copyright 1992, EDS; All rights reserved; Permission to copy must be explicitly granted.

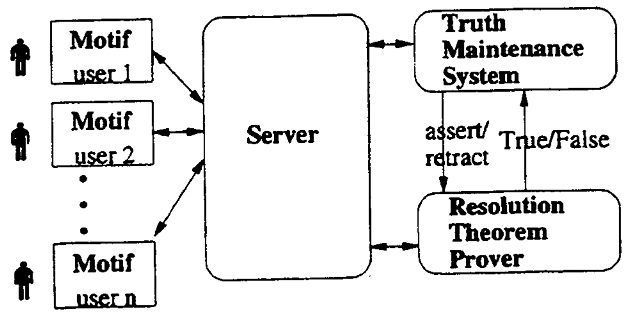
Benefits of Formalization

The basic goal is understanding the knowledge

- •Concise Notations
- •Assumptions
- •Consensus Communication
- •Consistency
- •Completeness
- •Inference
- •Framework for Extensibility

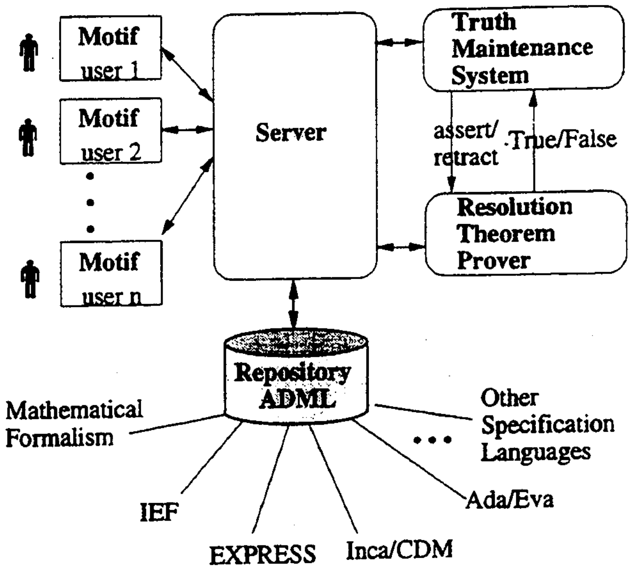
© Copyright 1992, EDS: All rights reserved; Permission to copy must be explicitly granted.

Consistency Is Maintained Using An Automated Theorem Prover

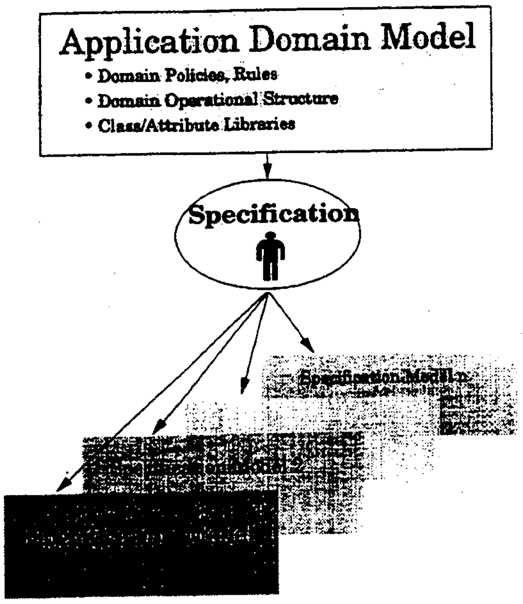


© Copyright 1992, EDS: All rights reserved; Permission to copy must be explicitly granted.

Specification Models Can Be Transformed Into Other Specification Languages



© Copyright 1992, EDS: All rights reserved; Permission to copy must be explicitly granted



© Copyright 1992, EDS: All rights reserved. Permission to copy must be explicitly granted.

Instantiating Specification Models from a Domain Model

- **Applicable attributes**

```
(applicable Class Attr) →
     ∃spec_model(used spec_model Class Attr)
```

- **Must_have attributes**

```
(must_have Class Attr) →
     ∀spec_model(used spec_model Class Attr)
```

© Copyright 1992. EDS: All rights reserved: Permission to copy must be explicitly granted.

Instantiating Specification Models from a Domain Model

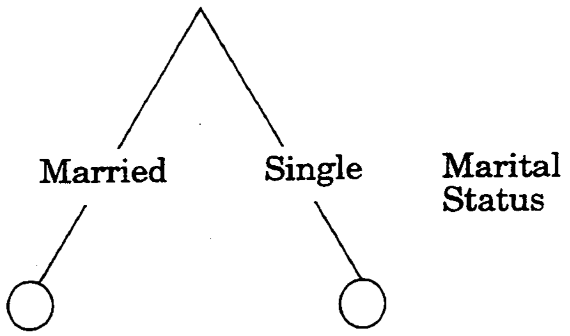
- **Conditional applicable attributes.**

```
(cond_applicable Class Attr (P (a Class) val)†)
  →∀spec_model
     [(used spec_model (P (a Class) val) Attr)
       →(used spec_model Class a)]
```

† For example, `(= (Gender Person) Female)` is interpreted as the subclass of `Person` within which the value of attribute `Gender` is `Female`.

© Copyright 1992. EDS. All rights reserved. Permission to copy must be explicitly granted.

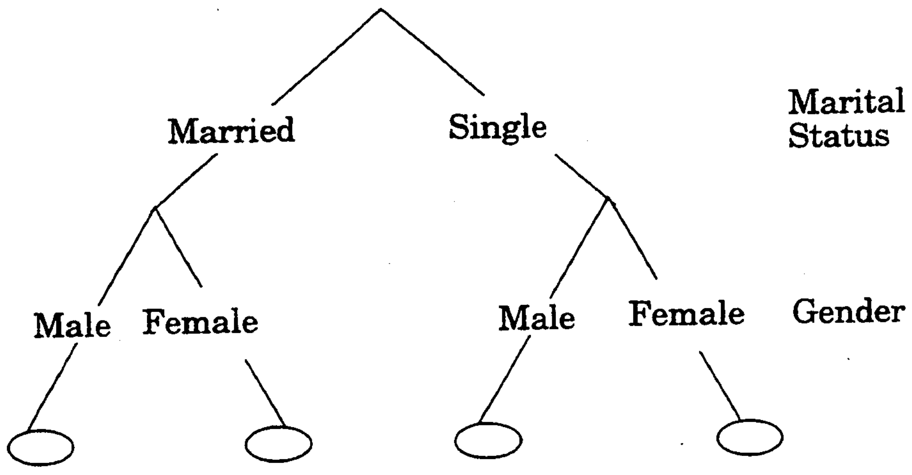
Maternity Benefit $



```
(applicable Person Maternity_benefit$)
```

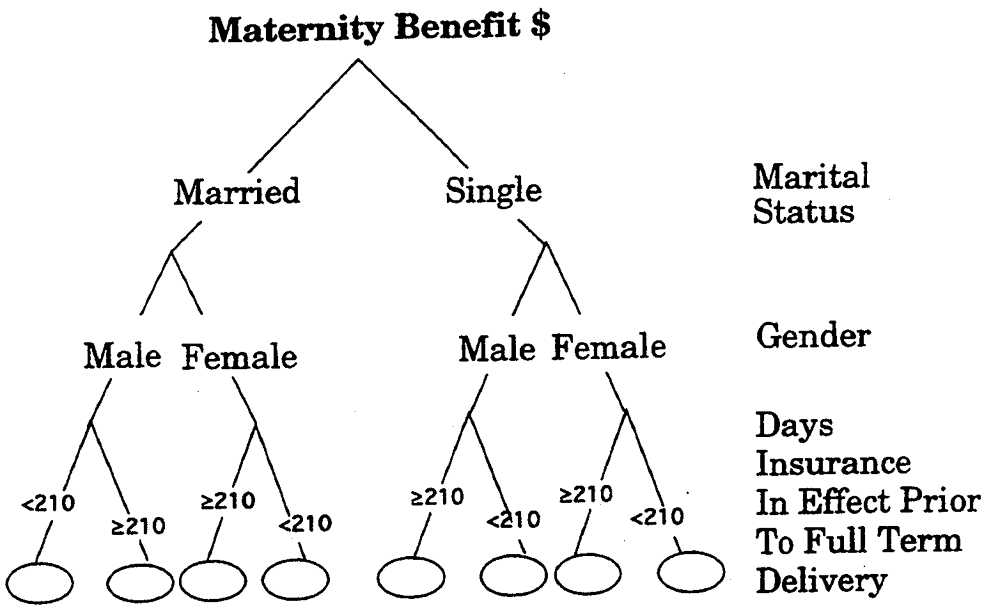
© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted.

Maternity Benefit $



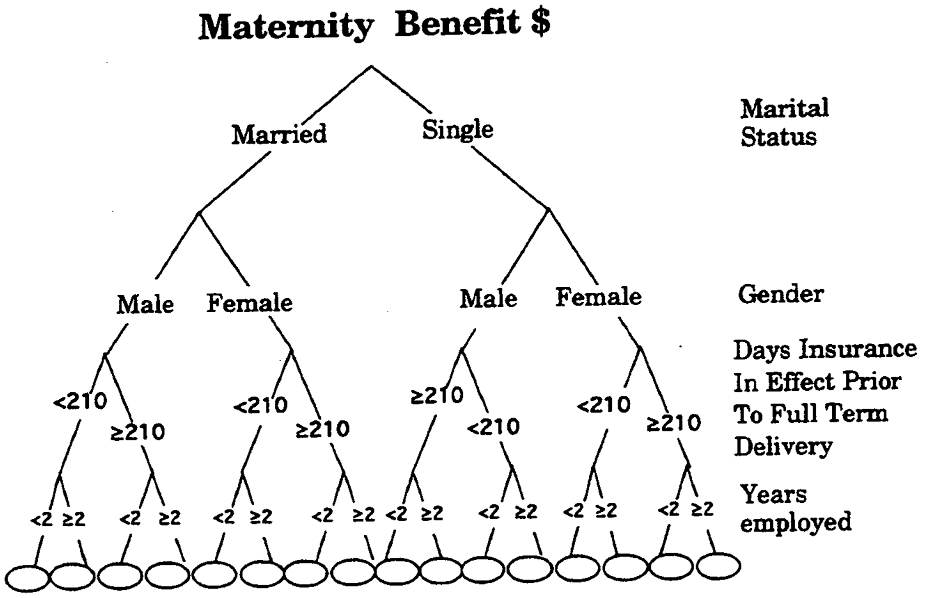
```
(cond_applicable Person Maternity_benefit$
     (= (Gender Person) Female)
```

© Copyright 1992. EDS. All rights reserved; Permission to copy must be explicitly granted



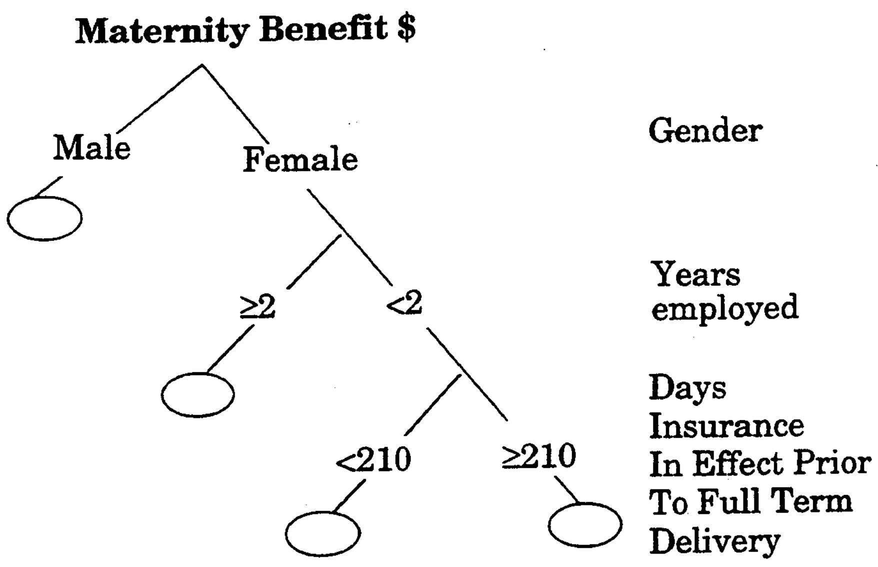
```
(cond_applicable Person Maternity_benefit$
 (& (= (Gender Person) Female)
    (≥ (DIPFT Person) 210)))
```

© Copyright 1992. EDS: All rights reserved: Permission to copy must be explicitly granted.



```
(cond_applicable Person Maternity_benefit$
 (& (= (Gender Person) Female)
    (OR (≥ (DIPFT Person) 210)
        (≥ (Years_employed Person) 2))))
```

© Copyright 1992. EDS: All rights reserved: Permission to copy must be explicitly granted.



```
(cond_applicable Person Maternity_benefit$
 (& (= (Gender Person) Female)
    (OR (≥ (DIPFT Person) 210)
        (≥ (Years_employed Person) 2))))
```

© Copyright 1992, EDS. All rights reserved: Permission to copy must be explicitly granted

Example: Maternity Benefit$

Domain Model:

```
(applicable Person Marital_status)
(applicable Person Gender)
(applicable Person Years_employed)
(cond_required Person
   Maternity_benefit$
       (& (= (Gender Person) Female)
        (OR (≥ (DIPFT Person) 210)
            (≥ (Years_employed Person) 2))))
```

© Copyright 1992, EDS. All rights reserved. Permission to copy must be explicitly granted