SANDIA REPORT

SAND2003-0804
Unlimited Release
Printed March 2003

Identifying and Implementing Patterns in Data Models

Shelley M. Eaton, Gregory N. Conrad

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia is a multiprogram laboratory operated by Sandia Corporation,
a Lockheed Martin Company, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865)576-8401
Facsimile: (865)576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.doe.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800)553-6847
Facsimile: (703)605-6900
E-Mail: orders@ntis.fedworld.gov
Online order: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2003-0804
Unlimited Release
Printed March 2003

Identifying and Implementing Patterns in Data Models

Shelley M. Eaton
Software and Information Engineering Department

Gregory N. Conrad
Advanced Decision Support Applications Department

Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1137

Abstract

This report describes a methodology to recognize data patterns, create generalized data models, and implement the resulting relational database structures. The processes that allow the data model patterns to emerge are described in detail so that a data modeler or designer can create a generalized data model, beginning with the validated and verified literal data model from the customer's point of view. In addition, this report discusses the value of generalization and effective methods to employ for implementing generalized data models, showing that generalization is a powerful method to handle changing or unknown requirements. Although requiring an investment for transitioning understanding to the developers and additional code to implement the database structure, a generalized data model can provide the optimal solution for the software product's data structure in its initial phases and perhaps its entire life.

Acknowledgements

The authors thank the following people for their creativity, support, and vision for information modeling, creating generalized relational database structures, and devising effective methods to implement those generalized database structures: Michael K. Bencoe, 9519; Olin H. Bray, 6544; David S. Cuyler, 9514; and Michael J. Eaton, 5852.

Table of Contents

Acronyms

ER	Entity Relationship
ORM	Object-Role Modeling
UoD	Universe of Discourse

Identifying and Implementing Patterns in Data Models

Introduction

As information systems become more complex and new technologies are rapidly created to aid in the software development process, software professionals continue to seek new methodologies to increase the success rate in their software development efforts. It seems as though new or reconditioned software techniques, products, and methodologies are introduced yearly as the solution for software development problems. And though many of these silver bullets eventually fall by the wayside, some techniques will continue to be reliable and successful, such as identifying and implementing data patterns to create robust and flexible data structures.

For many software applications, especially data-intensive ones, the data structure forms a foundation for the software design and paves the way for further generalization in the interface and implementation designs. If the data structure is sound and flexible, the software development effort has a greater chance of progressing smoothly, as compared with the frustrations involved with having to "code around" a faulty data structure or to continually modify the data structure and the supporting code due to changing requirements. And while changes to requirements are expected, a generalized data structure can lessen the impact of implementing those changes.

> "If we can model in this sense, using simpler and more generic models, we will find they stand the test of time better, are cheaper to implement and maintain, and often cater to changes in the business not known about initially. For example, rather than just model the exact organisation structure in our business, we could use a generic organisation model that can accommodate executive change (often just whim!). The generic model should even be able to handle the acquisition of another company or a merger with another department—without changing the implementation design. It should be possible simply to declare the new structure to the system." (Hay 1996, page xvii)

Identifying patterns in data relationships and how data are used forms the foundation to establishing a generalized data structure, providing built-in flexibility to handle changing or unknown requirements. In *Problem Frames, Analyzing and Structuring Software Development Problems*, Michael Jackson states that "The proper response to uncertainty and fluidity of requirements is not to throw up your hands and abandon the task. It is to work at a higher level of generality." (Jackson 2001, page xvi) Generalizing and classifying also helps manage enormous or complex sets of data, relationships, and constraints. "Classification is one mechanism we use to manage the complexity of the objects in the world," according to Martin and Odell. (Martin and Odell 1998, page 75)

The key to building a generalized data structure begins with problem analysis and domain characterization, where initial data modeling occurs and data patterns first begin to emerge. C. J. Date says "...that the data model is *crucial* and *central* and *fundamental*..." (Date 2000, page 24) "The data

model is a representation of the things of significance to an enterprise and the relationships among those things. It portrays the underlying structure of the enterprise's data..." (Hay 1996, page 1)

The modeler, designer, and implementor need to be vigilant to recognize patterns throughout the software life cycle, even after the information system is put into production. Some patterns could be recognized in initial modeling sessions, while others may emerge much later on in the design and implementation efforts. Identifying patterns provides one of the best reuse benefits in software development.

The purpose of this document is to answer three main questions: 1) How do you transition a literal data model to a generalized model? 2) Is there an optimal generalized data model? and 3) What are the issues in implementing a generalized data model? The process steps outlined in this report are very detailed for the purpose of understanding by modelers and developers who have never worked with generalized data models. As people gain experience with this methodology, they will become more proficient in identifying opportunities for generalization and may even want to eliminate some of the process steps. This methodology incorporates verification and validation concepts throughout; therefore, you may be able to do some of the steps very quickly, but all are needed to ensure a complete analysis and a quality product.

This methodology focuses on relational database structures, illustrating the concepts with Object-Role Modeling (ORM), along with Entity Relationship (ER) diagrams. This methodology can be scaled for any size of project or any degree of complexity. It can also be used with various software development models, such as agile and waterfall. The example used to explain the basic process is a Lessons Learned database application.

Data Model Generalization

Emerging Data Patterns

The opportunity to identify patterns occurs throughout the entire software engineering life cycle. Patterns begin to emerge with the initial information gathering in the problem definition and domain characterization activities and continues through design and implementation. Typically numerous concrete examples of data and how that data are used comes from these initial discussions with the domain experts. These concrete examples are then studied to form an initial elementary generalization or classification.

> "Generalization enables us to examine whether these concepts have anything in common. Does a *more general* concept encompass concepts such as Shoe, Slacks, and Shirt? In general, these are known as Clothing. Clothing, in turn, might be encompassed by an even more general category called Merchandise or Household Article—and so on." (Martin and Odell 1998, page 76)

Even at this elementary level of generalization, it becomes clear that the data modeler's goal is not to identify an exhaustive list of examples to ensure a complete and accurate data model, but to identify a significant sampling of currently known examples in order to establish categories or ORM elementary data relationships, called fact types. This initial generalization of examples, also referred to as the literal

data model, can then be further abstracted to accommodate future incorporation of unknown or changeable data without any changes to the data structure. This expanded abstracted data model is known as the generalized data model. David C. Hay, in his book, *Data Model Patterns: Conventions of Thought*, expresses this concept in the following statement:

> "Entity modelling, or data modelling as it is sometimes called, may be used as a passive way of modelling exactly what exists—providing little interpretation or insight as to its meaning. There is a more active form of modelling, however, commonly found in mathematics and science, which has a model predict something that was not previously known or provide for some circumstance that does not yet exist. Such models are invariably much simpler, easier to understand, and yet deal with more situations than mirror-image models." (Hay 1996, page xvii)

Benefits of Generalization

A key benefit of a generalized database structure is that it is flexible and robust enough to accommodate new or changing data requirements. Changes are made at the row level in a relational database table, rather than modifying the tables by adding more columns or creating additional tables. There are also fewer relational tables and columns, lessening or eliminating data redundancy in the database structure. Generalization can be the salvation for a struggling project due to the customer's changing or unknown data requirements. If implementing a generalized model, progress on software development can continue even though the data structure is not fully populated.

Generalization also permits specified constraints to be encoded in the database, thereby providing flexibility for changing these constraints. Instead of writing code to enforce individual constraints, the population of the database is updated, reducing software development and maintenance costs.

Generalization is akin to code reuse, but achieved much earlier in the software development life cycle. It begins in the requirements phase and continues in the design and architecture activities. By generalizing object types and data relationships, it sets the stage for further generalization, such as interface design.

In addition, the process of generalization can provide a deeper understanding of the problem, data, and data relationships. A superficial understanding can lead to a quick implementation, but one that is based on just a current view of the problem. Generalization can force a person to more fully examine the data and data relationships, revealing a more profound understanding of the problem being addressed.

Costs of Implementing Generalized Data Structures

This generalization does not come without cost, however. Criteria to be considered when generalizing include the probability and risk of changing requirements; the various ways the data will be used; the difficulty of implementation, maintenance, and enhancements; and the thinking styles of the project team.

A generalized data structure can be seductive, especially to people with meta thinking styles, because you can accommodate any kind of data in a more elegant fashion. Theoretically, a generalized model can be considered the "best" model because it provide more flexibility. However, the data constraints identified in the literal model can no longer be enforced in the database; instead, they are enforced in the process

layer, which will require a more robust design and implementation. Identifying and enforcing constraints is key for having verified data. This implementation will initially cost more to develop.

A generalized data structure also produces a problem for transactional data; e.g. inserting multiple types of data. Static types of data, like Person Gender, have an extremely low probability of changing. Static data can be accommodated in a generalized data structure; but there is a cost of introducing ambiguity into the create, read, update, and delete processes.

In addition, abstract or meta-level thinkers will have an easier time understanding the generalized data structure, while the literal thinkers could struggle with comprehending what appears to be an unnecessarily complex data structure. And while one style of thinking is not necessarily better than the other, the team members' thinking styles need to be carefully considered when analyzing whether or not to employ generalized database structures in software development. In any event, it is crucial that all team members recognize people's thinking styles in relationship to literal or generalized concepts in order to avoid miscommunication.

When to Generalize

The question that needs to be answered during software development is not, "Should the database structure be generalized?" but instead, "At what point does generalizing the database structure fail to provide benefit for the cost involved over the life of the software product?" Each project is unique, and the cost/benefit factor must be analyzed independently. Additionally, the cost/benefit factor needs to be reanalyzed throughout the life of the software product. It is possible that some changeable data will eventually stabilize at some point in the product life cycle. It may be beneficial to remove the now stabilized data from a generalized structure to a literal structure and recode as necessary, increasing the understandability of the code and the database structure, while enforcing constraints in the database.

Regardless of the level of generalization chosen for implementation, the data model needs to be fully generalized in order to gain a robust understanding of the data and data relationships. Doing so will aid in determining the optimal data structure.

Developing the Data Models

While many software professionals agree that a generalized data structure is beneficial, they often approach the process in an ad hoc manner with little thought to verification and validation. The methodology illustrated in this report integrates verification and validation activities, from initial conceptual data model creation through generalization. And while some modelers postpone generalizing the data model to a future release of the software, requiring considerable effort to implement, this methodology makes that delay unnecessary. Generalization may be completed in the early iterations of software development life cycle or may be realized in later iterations.

Object Role Modeling (ORM) Data Modeling

The conceptual (literal) data model, describes the structure that is relevant to the domain being addressed, sometimes called the Universe of Discourse, which is documented in a formal schema. This literal model illustrates the structure from the perspective of the domain expert. According to Terry Halpin in *Information Modeling and Relational Databases*, "ORM is a method for modeling and querying an information system at the conceptual level, and for mapping between conceptual and logical levels." (Halpin 2001, page 56)

The ORM data modeling methodology is advantageous over others, such as the Entity/Relationship methodology, because many integrity constraints can be included in the model, whether represented in textual or graphical notation. "Although E/R diagrams can be useful for explicating the structure of the database at a high intuitive level...the trouble is that they're virtually incapable of representing integrity constraints (except for a few special cases, including foreign key constraints in particular...). And as far as I'm concerned, by contrast, database design is really all about specifying integrity constraints! Database design **is** constraint definition." (Date 2000, page 115)

The literal data model documents the communication between the domain expert and the data modeler at the most elementary level, using elementary facts. "...An elementary fact is a simple assertion, or atomic proposition, about the UoD." (Halpin 2001, page 60) "Elementary facts are assertions that particular objects play particular roles." (Halpin 2001, page 61) A fact type is of a set of generalized elementary facts.

This literal data model is the starting point for the entire generalization process. The domain expert verifies and validates the literal model. Because the literal model will eventually be generalized to a point where the domain expert may no longer understand its information content, the verified and validated literal model is a critical component in creating the generalized model.

Terry Halpin lists the following seven steps in creating the literal data model in *Information Modeling and Relational Databases:* (Halpin 2001, page 59)

1. Transform familiar information examples into elementary facts, and apply quality checks.
2. Draw the fact types, and apply a population check.
3. Check for entity types that should be combined, and note any arithmetic derivations.
4. Add uniqueness constraints, and check arity of fact types. [unary, binary, ternary]
5. Add mandatory role constraints, and check for logical derivations.
6. Add value, set comparison, and subtyping constraints.
7. Add other constraints and perform final checks.

This same process is also employed to create the generalized model, populated by the literal models fact types and populated examples.

Lessons Learned Database Application Example

The example used in this report is a very simple Lessons Learned database application that introduces the generalization process. After initial discussions with the Lessons Learned customers, it was decided to also include Preferred Practices in the same database. This example includes sample discussions with the domain experts, the resulting ORM and ER models, and explanations of how changing requirements can alter those models. Verification and validation activities are also included.

Note: The example data models shown in this paper represent just a portion of a complete data model. Consequently, the example data models do not necessarily contain complete and accurate notation. The reader's intent is not to complete the full data model or to correct notation, but to use the example models to help understand the points being made by the authors. In addition, Object-Role Modeling (ORM) is not fully explained. Additional information on ORM can be found in Terry Halpin's *Information Modeling and Relational Databases* and at the www.orm.net website.

Step 1: Create the Literal Data Model

The literal data model begins to take shape through discussions with the domain experts about the problem that needs to be solved and from characterizing the domain. Questions to the domain experts might include:

- What is a Lesson Learned?
- What do you want to know about a Lesson Learned?
- What are some examples of Lessons Learned?
- How would you use Lessons Learned information?
- Who is interested in knowing about Lessons Learned?

The domain experts can answer with concrete examples, facts, categories of information, definitions, or generalized statements (fact types). Information can also be obtained through observations of the domain expert's environment, users performing work, and examination of documentation. These are all useful for sketching out the fact types and populating those fact types with concrete examples in the initial version of the literal data model. The modeler is interested in this information to:

- Gain a common understanding of the problem being solved
- Ensure that fact types are elementary
- Validate that the fact types represents the problem being addressed
- Verify that the documented fact types are correct and complete, from the domain expert's point of view, using concrete examples
- Understand the data's relationships and constraints
- Determine the data type and length of the objects by examination of the concrete examples

The following is an excerpt of a sample conversation between a Lessons Learned domain expert and the data modeler. The domain expert is explaining an example of a lesson learned to the data modeler.

> One lesson I learned is that we need to design processes based on testing. Testing is the key. The advantage to this is that we can specify the testing process before the project begins. If we don't do test-based design, then typically we don't test for static protection and eventually people become frustrated and transfer out of the organization. However, the down-side of test-based design is that more analysis time needs to be scheduled, and that effort costs money, which means you have to budget for it ahead of time. That's not always possible with how we do business.

After hearing similar stories, the data modeler organizes the information into a formalized schema by generalizing the example facts into fact types, as shown in Table 1 on page 13. These natural language fact types are equivalent to the literal model's graphical notation as shown in Model 1 on page 14. The order of the fact types has no relevance. In addition, the following table shows only one example fact. Additional examples are documented in the graphical notation shown in Model 1 on page 14. A legend for the ORM graphical notation is located in Appendix C.

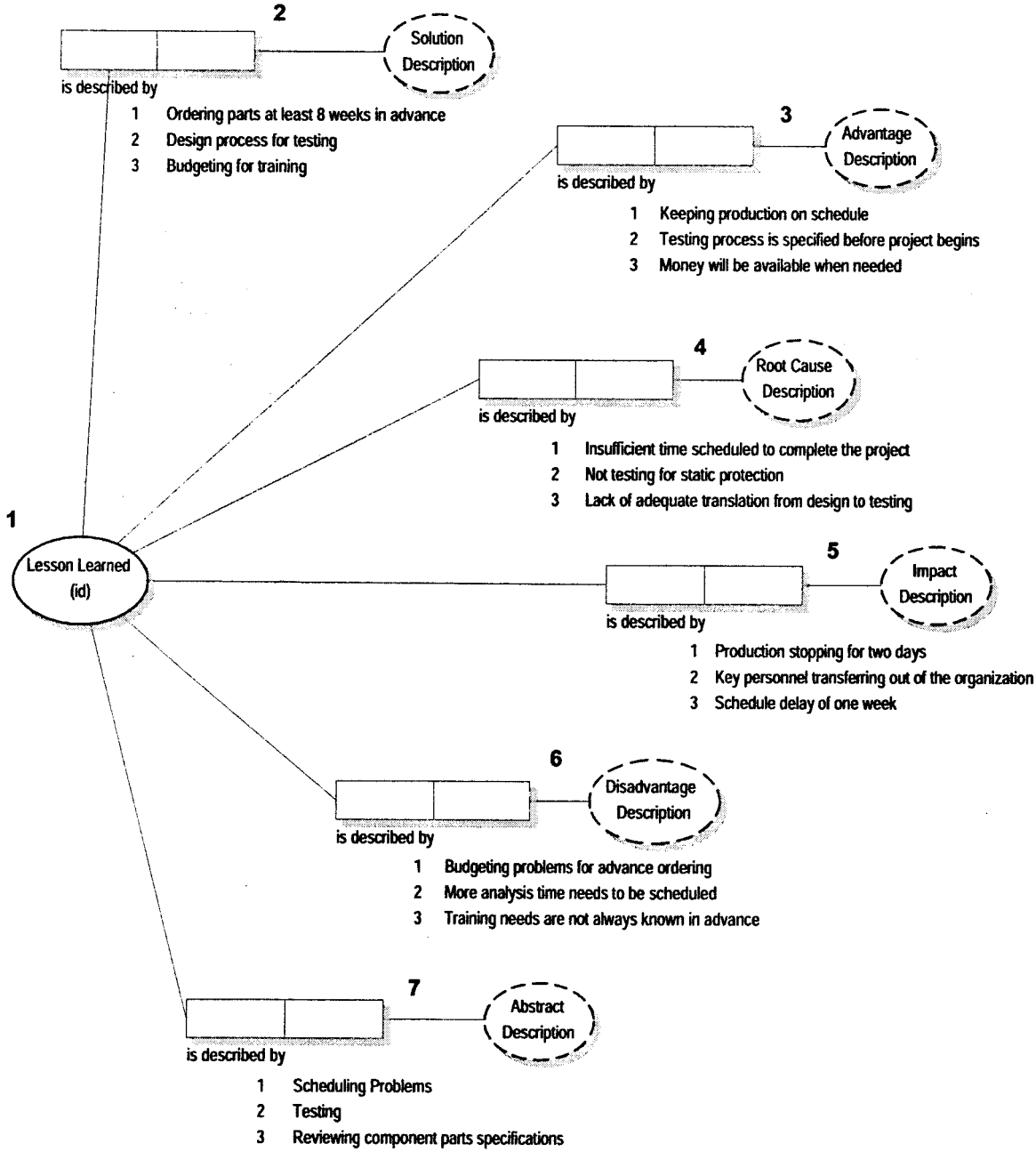
Table 1: Lessons Learned Example Fact Sentences and Resulting Fact Types

Communicated Example Facts	Resulting Literal Model Fact Types
1. This is the first example Lesson Learned.	A Lesson Learned exists and is uniquely identified.
2. The way we solved this problem was to design the process for testing.	A Lesson Learned is described by a Solution Description.
3. This solution was advantageous because the testing process is specified before the project begins.	A Lesson Learned is described by an Advantage Description.
4. The root cause for this problem was not testing for static protection.	A Lesson Learned is described by a Root Cause Description.
5. The impact of not designing the process for testing is that key personnel transfer out of the organization.	A Lesson Learned is described by an Impact Description.
6. A disadvantage for designing the testing process is that more analysis time needs to be scheduled which costs more and requires advanced planning.	A Lesson Learned is described by a Disadvantage Description.
7. This lesson learned deals with testing.	A Lesson Learned is described by an Abstract Description.

The literal data model graphical representation shown in Model 1 on page 14 is semantically equivalent to the natural language fact types in the second column of Data Table 1 on page 13; however, the focus is different. The textual representation allows the reader to concentrate on an one fact or fact type, while the graphical notation allows the reader to get a sense of the whole and the connections within the whole. Model 1 on page 14 also includes additional concrete examples, which help to verify the fact types.

The fact types in Model 1 on page 14 can be read in sentence format, as documented in the second column of Table 1 on page 13. For example, Fact Type #3 reads "A Lesson Learned is described by an Advantage Description." You can also insert concrete examples into the Fact Type. The first example for Fact Type #3 would read "Lesson Learned #1 is described by an Advantage Description of "Keeping production on schedule."

Model 1: Lessons Learned Literal Data Model Without Constraints



Step 2: Determine Constraints in the Literal Data Model

After determining the literal data model's fact types, the concrete examples are used to establish the uniqueness and mandatory constraints for each fact type, which are then verified by the domain expert.

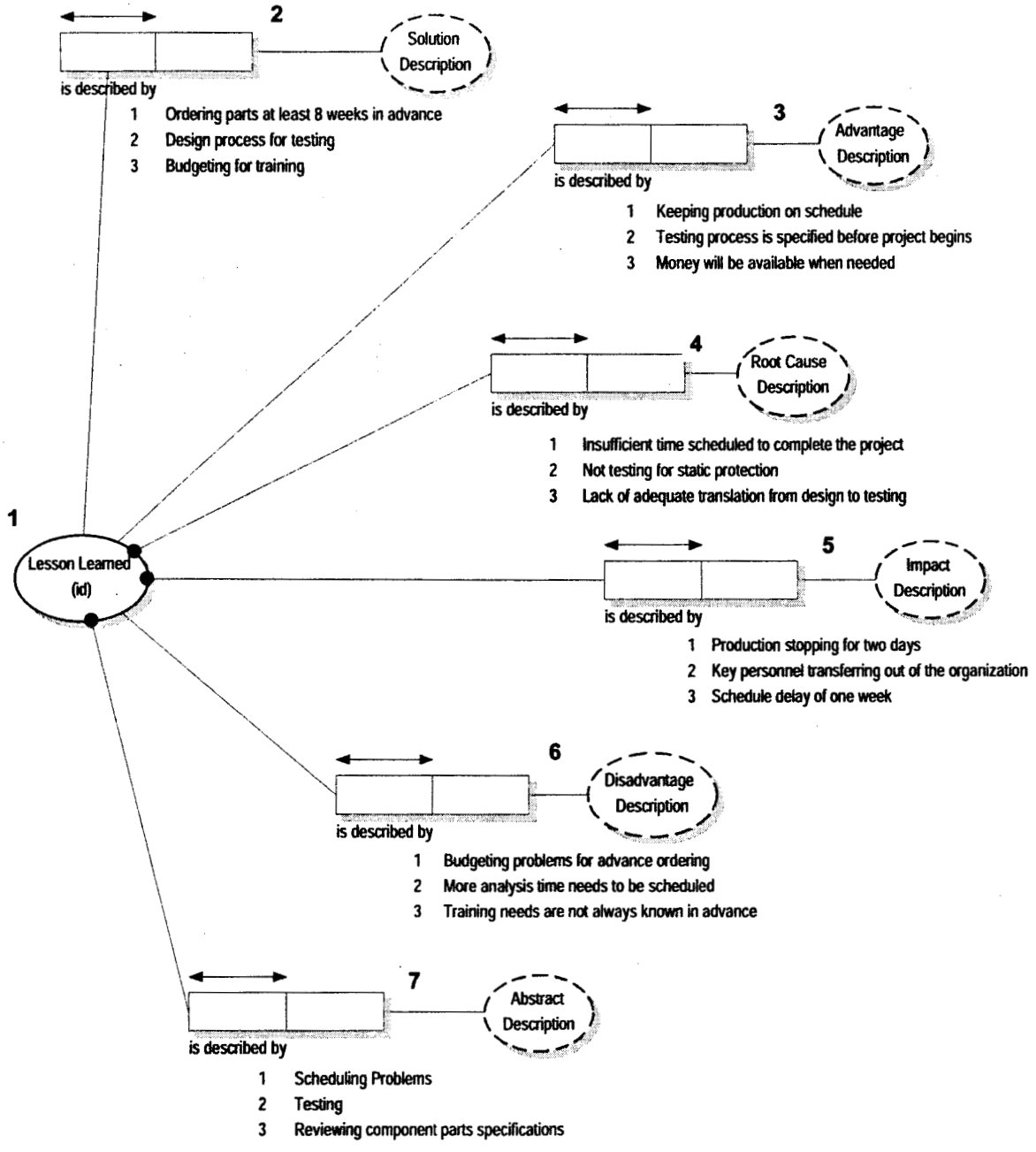
The mandatory constraints may be dependent upon state; however, for purposes of illustration in this report, state is not a consideration for the data models and resulting tables.

Table 2 on page 15 lists the seven fact types shown in Model 1 on page 14, along with the resulting constrained fact types. The resulting graphical notation follows in Model 2 on page 16.

Table 2: Lessons Learned Fact Types and Resulting Constrained Fact Types

Fact Types	Resulting Constrained Fact Types
1. A Lesson Learned exists and is uniquely identified.	A Lesson Learned exists and must be identified by one unique ID.
2. A Lesson Learned is described by a Solution Description.	A Lesson Learned may be described by at most one Solution Description.
3. A Lesson Learned is described by an Advantage Description.	A Lesson Learned may be described by at most one Advantage Description.
4. A Lesson Learned is described by a Root Cause Description.	A Lesson Learned must be described by exactly one Root Cause Description.
5. A Lesson Learned is described by an Impact Description.	A Lesson Learned must be described by exactly one Impact Description.
6. A Lesson Learned is described by a Disadvantage Description.	A Lesson Learned may be described by at most one Disadvantage Description.
7. A Lesson Learned is described by an Abstract Description.	A Lesson Learned must be described by exactly one Abstract Description.

Model 2: Lessons Learned Literal Data Model With Constraints



These seven fact types result in a single relational database table, as shown in Data Table 1 on page 17.

Data Table 1: Resulting Lessons Learned Literal Model's Relational Database Table

Lesson Learned		
PK	Lesson Learned id	Integer
	Abstract Description	Char(255)
	Root Cause Description	Char(255)
	Impact Description	Char(255)
	Solution Description	Char(255)
	Advantage Description	Char(255)
	Disadvantage Description	Char(255)

Step 3: Modifications to the Initial Literal Data Model

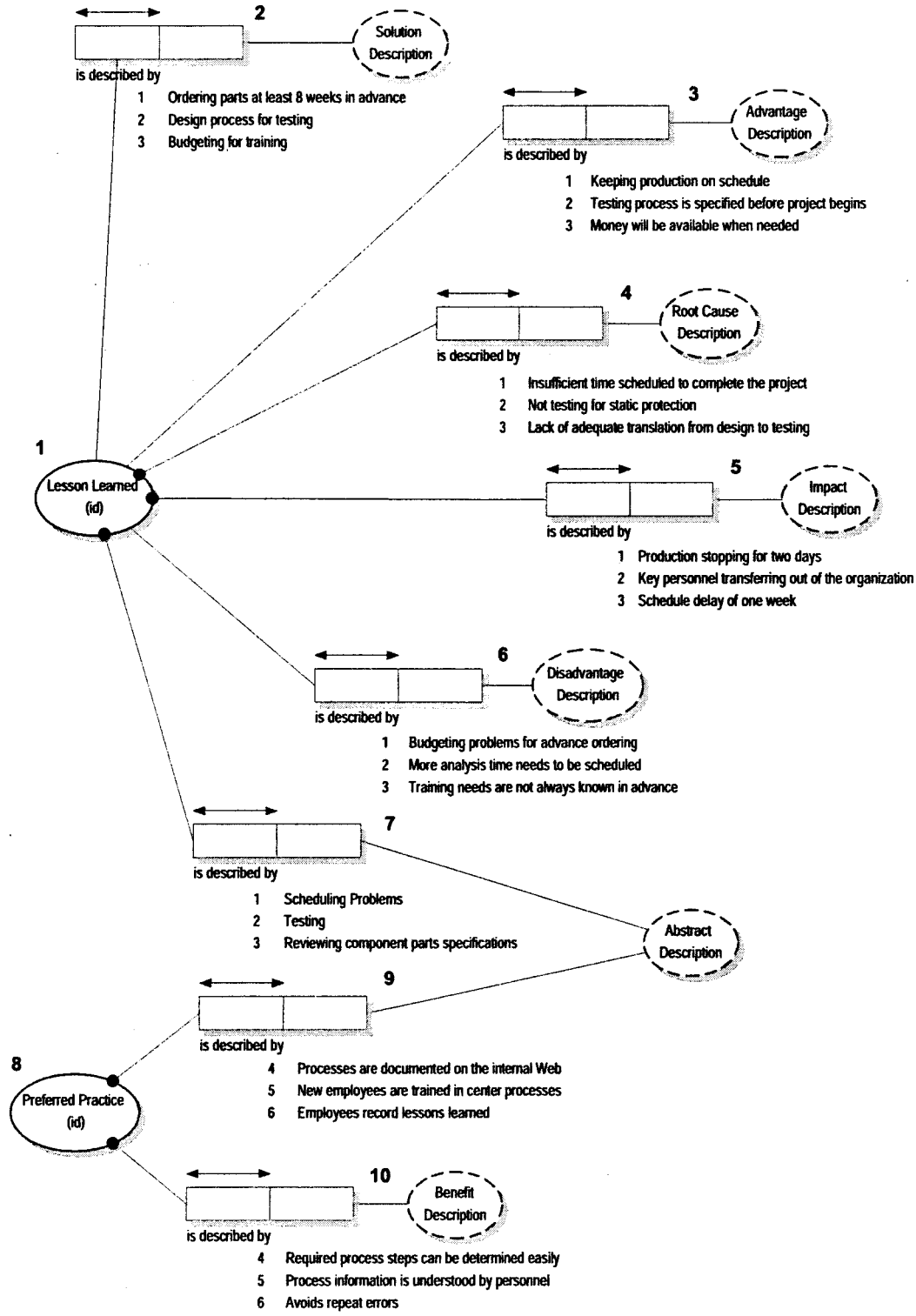
After the initial fact types are identified, changes and additions may be identified as new information becomes available or the domain expert gains new insights. In fact, the Lessons Learned customer in this example later decided to also include Preferred Practices in the same database application. After following the process for the initial Lessons Learned literal data model, additional constrained fact types were added, as listed in Table 3 on page 17.

Table 3: Additional Preferred Practices Constrained Fact Types

Additional Fact Types	Resulting Constrained Fact Types
8. A Preferred Practice exists and is uniquely identified.	A Preferred Practice exists and must be identified by one unique ID.
9. A Preferred Practice is described by an Abstract Description.	A Preferred Practice must be described by exactly one Abstract Description.
10. A Preferred Practice is described by a Benefit Description.	A Preferred Practice must be described by exactly one Benefit Description.

Model 3 on page 18 illustrates these additional Preferred Practices constrained fact types in the graphical notation, populated with examples.

Model 3: Lessons Learned and Preferred Practices Literal Model with Constraints



The result of adding these three Preferred Practices fact types (#8 - 10) is that an additional relational database table is created, thus changing the data structure.

Data Table 2: Resulting Lessons Learned and Preferred Practices Literal Model Database Tables

Lesson Learned		
PK	**Lesson Learned id**	Integer
	Abstract Description	Char(255)
	Root Cause Description	Char(255)
	Impact Description	Char(255)
	Solution Description	Char(255)
	Advantage Description	Char(255)
	Disadvantage Description	Char(255)

Preferred Practice		
PK	**Preferred Practice id**	Integer
	Abstract Description	Char(255)
	Benefit Description	Char(255)

As problem and domain characterization continues, requests for changes and additions are expected to occur. And while it is advantageous to create a literal model as accurately as possible and as early in the software engineering life cycle as possible, it is also realistic to expect that there will be modifications later on.

One goal during the initial modeling effort is to create a literal data model that is complete and accurate enough to eventually generalize. As a result of discussions with the domain expert and other field work, the data modeler may be able to predict or make knowledgeable judgments about what data could change or be added at a later date. These judgments can be incorporated in the generalized model to accommodate changing requirements.

Step 4: Verifying and Validating the Literal Data Model

One of the advantages of ORM is that it accommodates verification and validation by the domain expert. Since a literal data model is a data-focused representation of the communicated problem and the customer needs, the domain expert has the opportunity and the responsibility to verify that the fact types are correct by evaluating their populations. The domain expert is also able to validate if the fact types in the literal model accurately and completely address the problem to be solved.

One verification technique that can be employed is to map the literal data model's fact types to the processes, often represented as use cases. Is each fact type used by at least one process? If not, then either the fact type is extraneous or a process is missing. Do all the necessary fact types exist to support each process? If not, then either fact types are missing or the process is extraneous.

The domain expert is able to verify and validate the literal data model because it documents the information from the viewpoint of the domain expert, at the appropriate level of abstraction, and in a presentation style that the domain expert can readily understand. This is critical for the generalization process. If the domain expert has verified and validated the literal model, then the generalized model can be created on a firm foundation.

Step 5: Identifying Data Patterns in the Literal Model

During problem characterization and data modeling activities, data patterns begin to emerge from the domain expert's concrete examples and scenarios of how data are used. These examples can represent the current or the desired state of the domain being modeled; however, some unknown or predicted data may emerge. These unknowns and predictions may include data that need further investigation, domain understanding that is not possible at the current time, and fact types that will most likely change over time. All of this information, both known or unknown areas, are candidates for a generalized data structure.

The basic technique used to determine if the potential for data model generalization exists is to identify the similarities in the literal model, while noting the differences, including the following:

- Recognition of similar graphical notation in the model, such as uniqueness constraints and fact type arity (unary, binary, ternary)
- Study of the text notation for comparable meaning, including object labels and role verbs
- Comparison of similarities in how the data are used
- Identification of similar data types and lengths, based on the population of concrete examples
- Experimentation with the organization and layout of fact types in the graphical notation to discover alternate generalization solutions
- Recognition of previously identified patterns, based on past experience
- Awareness of overly complex relationships, involving numerous crossing lines and duplicate object types

The commonalities in the literal data model are not necessarily ordinarily recognized in the "real world," such as shoes and socks generalized as clothing. Often the commonalities involve how the data are used or how it behaves from an information system perspective. For instance, in the Need To Know Infrastructure generalized model, Rule, Piece of Information, and Person Group objects were generalized. At first glance, you may not think those three objects have any commonality. However, all three need to have their access controlled on a need-to-know basis; therefore, there is a high level of commonality amongst the three and are thus candidates for generalization.

In the Lessons Learned and Preferred Practice model (Model 3 on page 18), the commonalities are that the Lessons Learned and Preferred Practices concepts are described by Description Attributes, the fact types have the same uniqueness constraints, and the data lengths and types are the same. A strong pattern is visible in the graphical notation. If you held the printed copy of the data model at a distance where the text could not be read, the fact types all look the same. In addition, all of the Attribute Names end with the word "Description." The use of consistent naming conventions by the data modeler is beneficial in recognizing patterns.

Step 6: Subtyping the Literal Model

When the literal model is being formulated, the data modeler and designer can review the model looking for commonalities amongst the attributes and the objects for the purposes of subtyping. Subtyping offers several advantages over the literal model. It reduces the number of tables needed, avoids storing

redundant data, and allows for less significant modifications to the data structure by adding columns to existing tables instead of creating additional tables. In addition, the subtyping process can provide further insight to understanding the problem, the data, and the data relationships.

The two requirements to create a supertype with subtypes are: 1) at least one fact type must apply to all subtypes and 2) all the subtypes must have the same unique identifier. In the Lessons Learned example, there is at least one common fact type for both the Lessons Learned and Preferred Practices objects—Fact Types #7 and #9 as shown in Model 3 on page 18. It is possible for the unique identifier scheme to be the same for both Lessons Learned and Preferred Practices objects by employing surrogate keys, so subtyping analysis is feasible.

The domain expert may see Lessons Learned and Preferred Practices objects as totally different and would not agree to subtyping; and from their point of view, that is understandable. However, the data modeler conducts the subtyping analysis based on the verified and validated literal model, thereby not actively involving the domain expert in this activity. The data modeler views the data from a different point of view—an information viewpoint—while the domain expert views the data via the user interface or the business process.

Therefore, any generalized data model does not normally need to be presented to the domain expert. However, since subtyping can offer more insight in problem understanding, it may be beneficial to appropriately involve the domain expert. Generalization can be viewed as a distinct design activity or a synergistic effort between the data modeler and the designer.

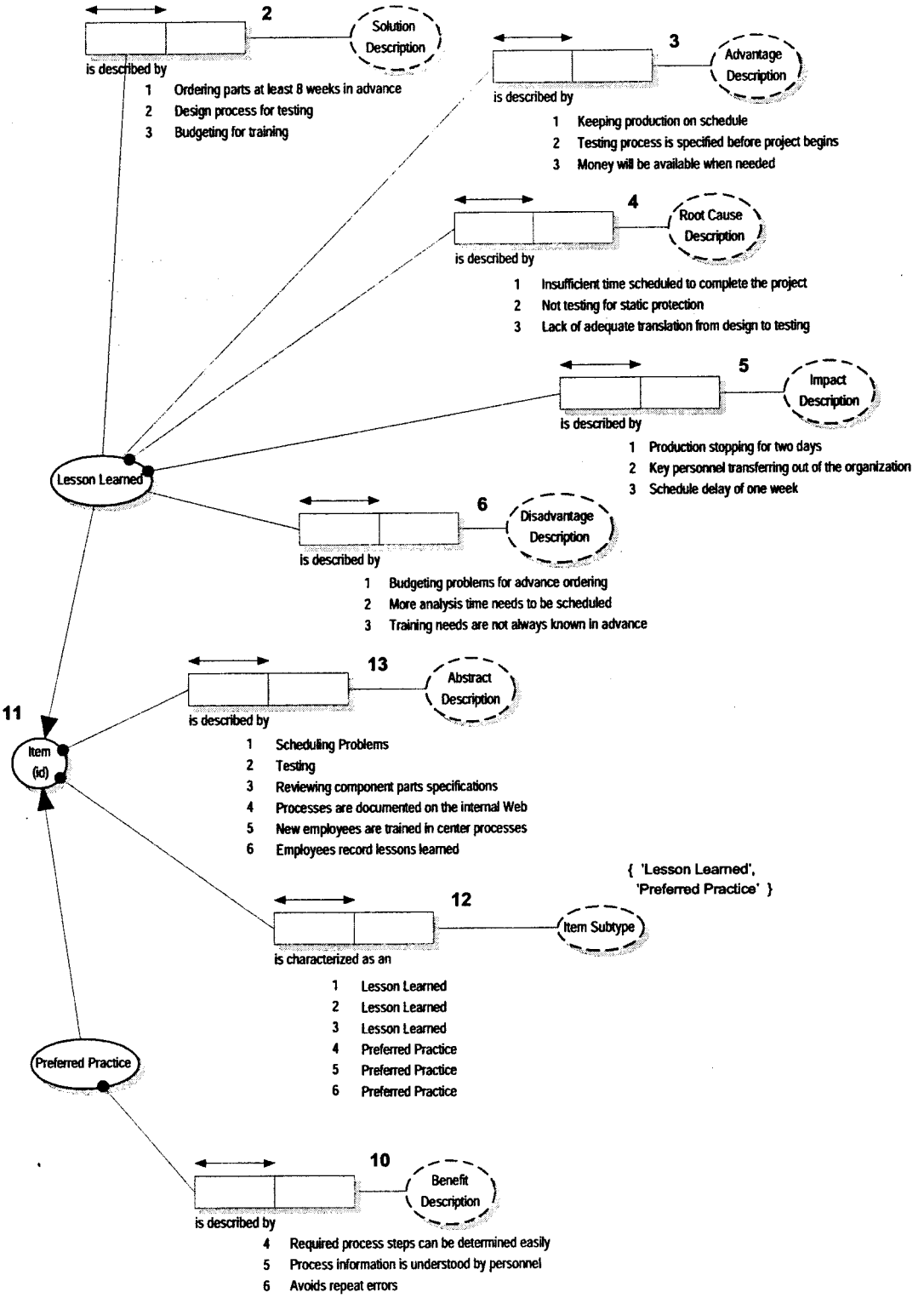
The Lessons Learned and Preferred Practice objects could become subtypes of a supertype called LL/PP or perhaps a more generic term, such as Item. Finding suitable generalized labels is often difficult, as there might not be an English word to accurately portray the commonality. After the subtyping analysis was completed, the fact types shown in Table 4 on page 21 emerged, which represents both Lessons Learned and Preferred Practices.

Table 4: Lessons Learned and Preferred Practices Suptyped Fact Types

Subtyped Fact Types
2. A Lesson Learned must have exactly one Solution Description.
3. A Lesson Learned may have at most one Advantage Description.
4. A Lesson Learned must have exactly one Root Cause Description.
5. A Lesson Learned must have exactly one Impact Description.
6. A Lesson Learned must have exactly one Disadvantage Description.
10. A Preferred Practice must have exactly one Benefit Description.
11. An Item exists and must be identified by one unique ID.
12. An Item must have exactly one Item Subtype (*values are Lesson Learned and Preferred Practice*).
13. An Item must be described by exactly one Abstract Description.

Lesson Learned and Preferred Practice objects now become subtypes of the supertype, Item. Fact #12 is added to distinguish between the subtypes. The ORM graphical notation is shown in Model 4 on page 23. The resulting single relational database table is shown in Data Table 3 on page 24. (Note that this single table implementation of subtyped data models is only one possible approach. A separate table for each subtype is just as valid.)

Model 4: Subtyped Lessons Learned and Preferred Practices Model



Data Table 3: Resulting Lessons Learned and Preferred Practices Subtyped Database Table

Item		
PK	Item id	Integer
	Abstract Description	Char(255)
	Root Cause Description	Char(255)
	Impact Description	Char(255)
	Solution Description	Char(255)
	Advantage Description	Char(255)
	Disadvantage Description	Char(255)
	Benefit Description	Char(255)
	Item Subtype	Char(25)

Subtyping Strengths and Issues

With this subtyped version of the Lessons Learned example, additional subtypes of Item could be accommodated by merely adding another value to Fact Type #12, which reads, "An Item must have exactly one Item Subtype." However, it is likely that additional columns would need to be added to the Item table to allow for additional attributes. And while this has minimal impact on the existing data structure, the structure will still need to be modified.

In addition, when a data model is subtyped, some integrity constraint information (also known as business rules) is lost, and the original non-subtyped literal model would still be needed for reference. Constraints would have to be handled in the process layer and not in the database.

Subtyping may help the modeler recognize some of the data patterns and can be thought of as an intermediary step in generalization. However, subtyping analysis is not absolutely necessary to proceed to the generalized model.

The Generalized Data Model: Grammar and Data Capture Fact Types

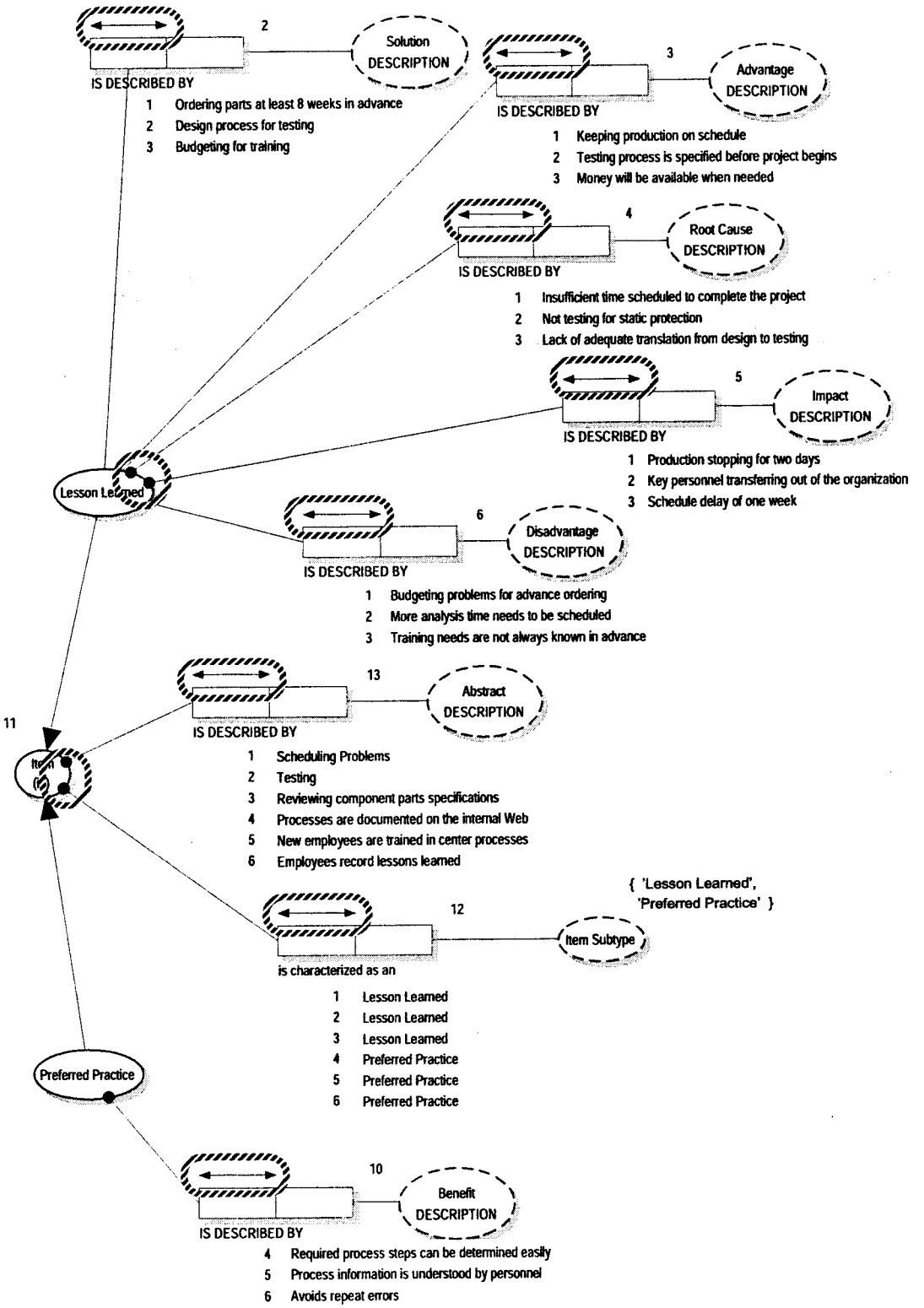
Subtyping does minimize the impact of changes to the data structure, but it does not allow explicit capture of the integrity constraints. However, there is a way to overcome these shortcomings. Generalization allows changes to be implemented at the row level of the relational database tables, instead of the column level, and is accomplished by creating a generalized data model consisting of both Grammar and Data Capture fact types.

The population of Grammar fact types control or allow certain values, constraints, or relationships to be captured in the Data Capture fact types. If these allowable values, constraints, or relationships change, only the Data Capture fact types' population changes. Thus, the Grammar fact types specify integrity constraints in the data population and not the data structure.

Grammar Fact Types

Our experience has indicated that a subtyped literal model can be easily transformed into a generalized model, as is true with the Lessons Learned model. Model 5 on page 25 is a duplicate of Model 4 on page 23, with similarities highlighted for potential generalization.

Model 5: Subtyped Lessons Learned and Preferred Practices Model with Similarities Highlighted



The literal model fact types are transformed into populated values of the generalized Grammar fact types, as shown in Table 5 on page 26. For instance, Item values convert to Item Type values and the various Description value objects convert to Descriptor Type values. The literal data model fact type uniqueness and mandatory constraints are also transformed into Grammar fact types. The Grammar can be thought of as a set of fact types that specify and enforce integrity constraints (or business rules), which are documented in the literal model. In fact, the generalized model is a meta model of the literal model. The following paragraphs describe the specific transformations from the literal Model 5 on page 25 to the Grammar Model 6 on page 27.

Grammar Fact Type #23 models the relationships between the Lessons Learned and Preferred Practice object types and the value object types (all of which are labeled with DESCRIPTION) in the literal model. The object type labeled "Descriptor Type" (#22) is created in the Grammar and populated with the literal model's value object type labels (i.e. Solution, Advantage, Root Cause, etc.). "Item Type" object type (#21) is also created in the Grammar and populated with the literal models object types (i.e. Lesson Learned, Preferred Practice). The role text in the literal model are changed from "IS DESCRIBED BY" to "allows description by" in Grammar model.

Grammar Fact Type #24 models the mandatory constraints of the "IS DESCRIBED BY" fact types in the literal model (encircled by dashed lines). The black dots next to the object types indicate that the roles are mandatory. If a literal model fact type has a black dot, then the population of Grammar Fact Type #24 is "Yes." If there is no black dot, then the population is "No."

Grammar Fact Type #25 models the uniqueness constraints of the "IS DESCRIBED BY" fact types (#2 - 6, 10, 12, and 13) in the literal model (encircled by dashed lines). The lines above the roles indicate the uniqueness constraint. If a literal model fact type has a line over the role closest to the Lesson Learned or Preferred Practice object type, then there is a M:1 relationship between the object type and its "DESCRIPTION" value. In other words, only one DESCRIPTION or answer is allowed; multiple DESCRIPTIONS or answers are not allowed. Thus, the population of Grammar Fact Type is "No." If any multiple answers are allowed, additional Grammar facts types would be necessary.

The Grammar fact types and population are verified against the literal model fact types and population. It is also possible to create Grammar fact types that control other constraints, such as values and data types.

Table 5: Lessons Learned Grammar Fact Types

Grammar Fact Types With Constraints
21. An Item Type exists and must be identified by one unique ID.
22. A Descriptor Type exists and must be identified by one unique ID.
23. Item Type may allow any number of descriptions by Descriptor Type. (Nominalized to Allowable Descriptor Type)
24. Allowable Descriptor Type is mandatory to populate? Y/N
25. Allowable Descriptor Type allows multiple entries? Y/N

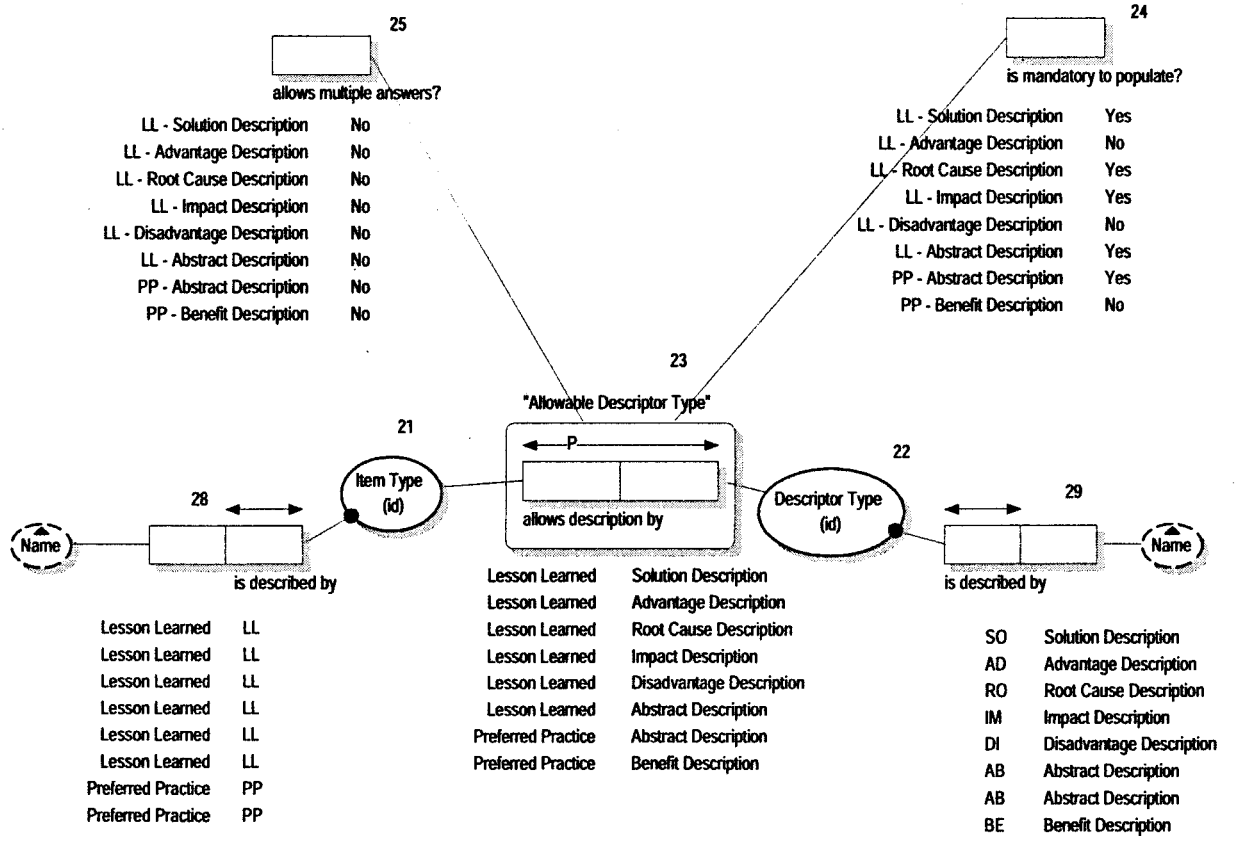
Table 5: Lessons Learned Grammar Fact Types

Grammar Fact Types With Constraints
28. Item Type must be described by exactly one Name.
29. Descriptor Type must be described by exactly one Name.

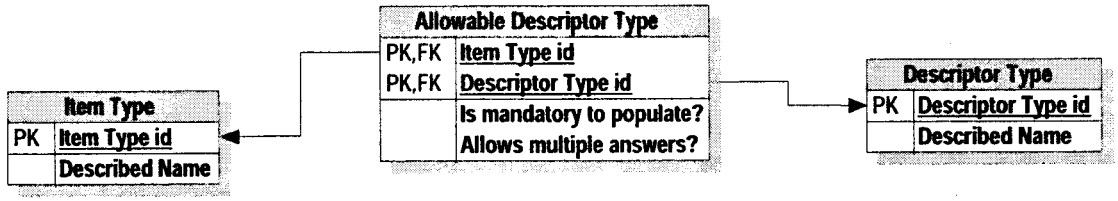
In this particular example, it was decided to relate the two unary fact types (fact types #24 and #25) to the nominalized object, Allowable Descriptor Type. However, it is possible to relate these fact types to the Descriptor Type object or to relate them to both objects. This depends on the value population. This subject is discussed further in the section entitled *Grammar To Control the Inverted Table.*

The Grammar graphical notation, shown in Model 6 on page 27, and resulting relational database tables, shown in Data Table 4 on page 28, follows.

Model 6: The Lessons Learned Grammar Generalized Model



Data Table 4: Resulting Grammar Database Tables



Data Capture Fact Types

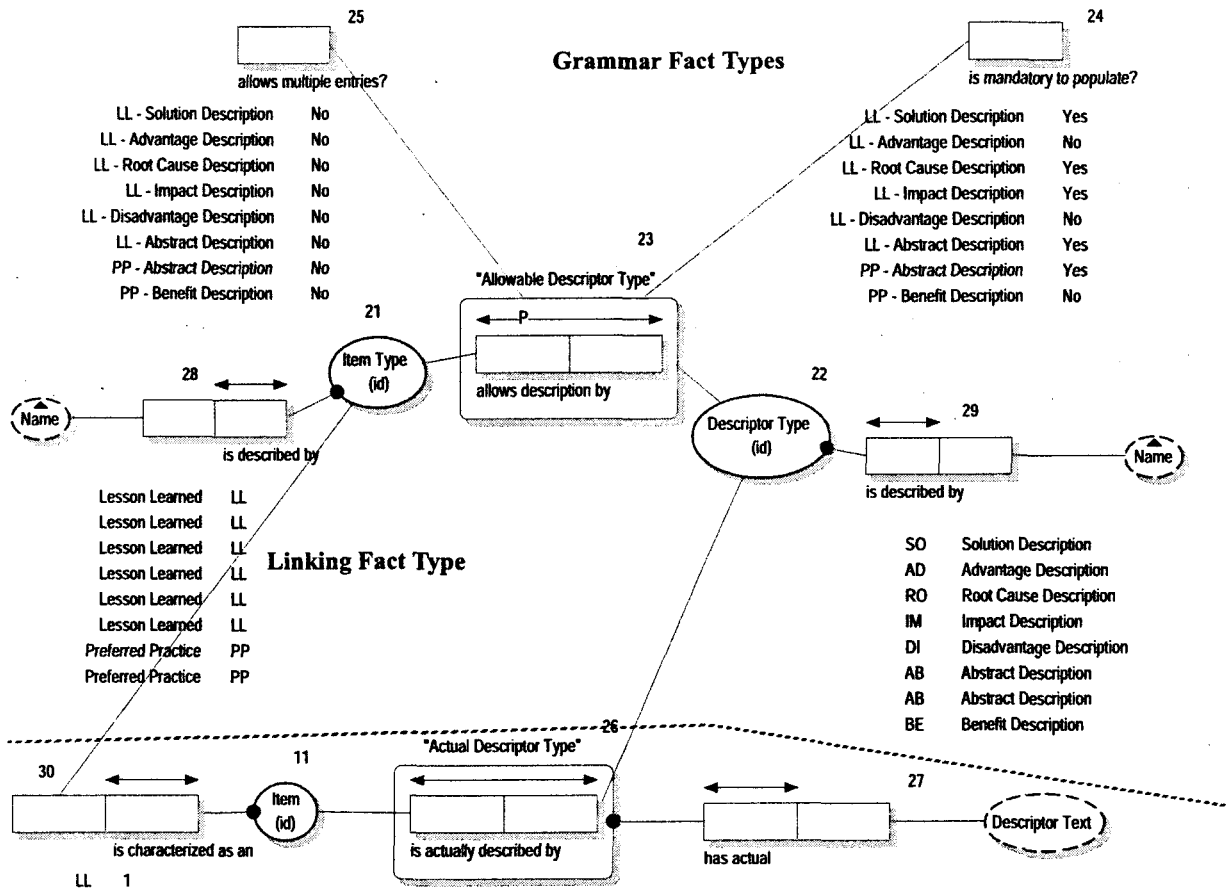
Fact types #26 and #27 actually capture the example data population that the literal model captured with both the Lessons Learned and Preferred Practices fact types #2-6, 10, 12, and 13, as shown in Model 4 on page 23. Fact type #30, "Item is of an Item Type," is a critical fact type, as it links the Data Capture fact types with the Grammar fact types. The Data Capture fact types are listed in Table 6 on page 28.

Table 6: Lessons Learned and Preferred Practices Data Capture Fact Types

Data Capture Fact Types With Constraints
30. Item is of an Item Type. (linking fact type)
26. Item may actually be described by any number of Description Types. (Nominalized to Actual Description Type)
27. Actual Description Type must be actually described by exactly one Description Text.

The Data Capture and the Grammar graphical notation, shown in Model 7 on page 29, and resulting relational database tables, shown in Data Table 5 on page 30, follows.

Model 7: Lessons Learned Grammar and Data Capture Generalized Model



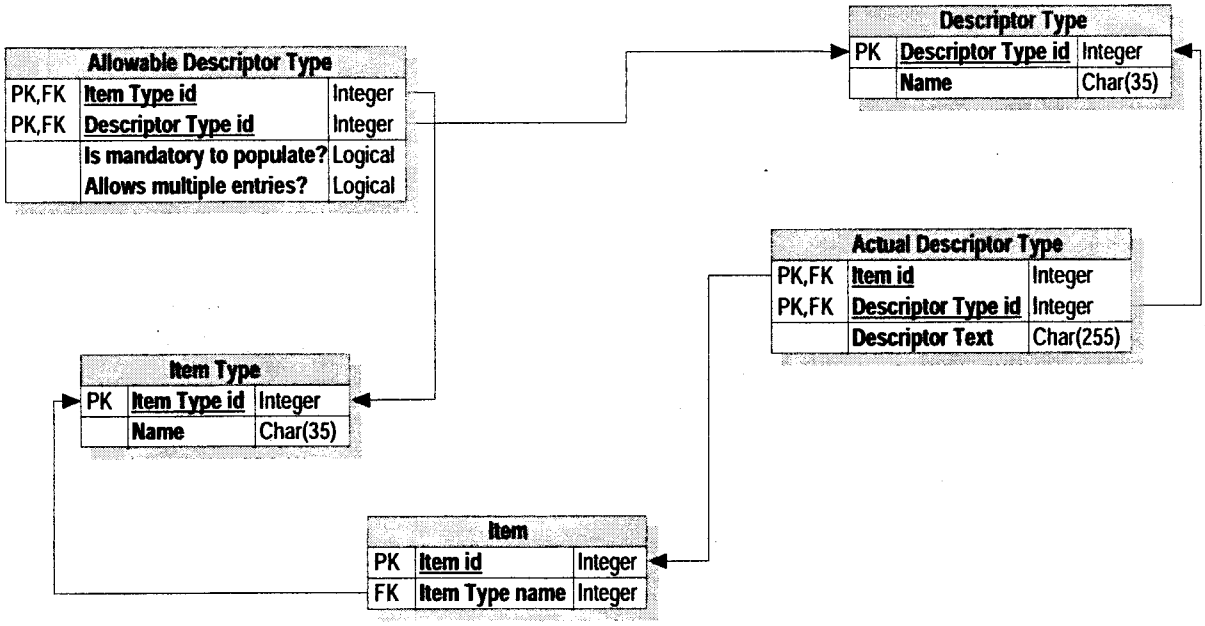
Data Capture Fact Types

LL	1
LL	2
LL	3
PP	4
PP	5
PP	6

1	Solution Desc
2	Solution Desc
3	Solution Desc
1	Advantage Desc
2	Advantage Desc
3	Advantage Desc
1	Root Cause Desc
2	Root Cause Desc
3	Root Cause Desc
1	Impact Desc
2	Impact Desc
3	Impact Desc
1	Disadvantage Desc
2	Disadvantage Desc
3	Disadvantage Desc
1	Abstract Desc
2	Abstract Desc
3	Abstract Desc
4	Abstract Desc
5	Abstract Desc
6	Abstract Desc
4	Benefit Description
5	Benefit Description
6	Benefit Description

1 - Solution Desc	Ordering parts at least 8 weeks in advance
2 - Solution Desc	Design process for testing
3 - Solution Desc	Budgeting for training
1 - Advantage Desc	Keeping production on schedule
2 - Advantage Desc	Testing process is specified before project begins
3 - Advantage Desc	Money will be available when needed
1 - Root Cause Desc	Insufficient time scheduled to complete the project
2 - Root Cause Desc	Not testing for static protection
3 - Root Cause Desc	Lack of adequate translation from design to testing
1 - Impact Desc	Production stopping for two days
2 - Impact Desc	Key personnel transferring out of the organization
3 - Impact Desc	Schedule delay of one week
1 - Disadvantage Desc	Budgeting problems for advance ordering
2 - Disadvantage Desc	More analysis time needs to be scheduled
3 - Disadvantage Desc	Training needs are not always known in advance
1 - Abstract Desc	Scheduling Problems
2 - Abstract Desc	Testing
3 - Abstract Desc	Reviewing component parts specifications
4 - Abstract Desc	Processes are documented on the internal Web
5 - Abstract Desc	New employees are trained in center processes
6 - Abstract Desc	Employees record lessons learned
4 - Benefit Desc	Required process steps can be determined easily
5 - Benefit Desc	Process information is understood by personnel
6 - Benefit Desc	Avoids repeat errors

Data Table 5: Resulting Grammar and Data Capture Database Tables



Data Modeling Summary

Three data models have been created as a result of the generalization process:

- Literal model, the lowest level of generalization that is domain specific
- Subtyped model, which is somewhat generalized from the literal model but is also directly tied to the domain
- Generalized model, consisting of the Grammar and Data Capture fact types, which is a leap from either the literal or subtyped models and is a non-domain specific, reusable product

All three models are equivalent in that the same data can be captured. While all three are implementable, each implementation would be greatly different. The generalized version can be viewed as a core product or a generic tool that can be tailored to a specific domain and one that forces a specific implementation design, while the others are tied to a specific application and its domain.

Implementation of a Generalized Data Structure

A generalized data structure was defined in a project for the Plutonium Disposition application. It is an extreme example of the data structures developed for the Lessons Learned example application discussed earlier in this report and is similar to Model 10 on page 47 in Appendix A. The Plutonium Disposition generalized model demonstrates the essential elements of a grammar-defined data capture mechanism. The data capture portion of the structure is an inverted table named Recorded Data, shown on Data Table 8 on page 34. By inverted it is meant that attributes are captured as rows in the table rather than

columns. For comparison, an example of a non-inverted or horizontally defined table is shown in Data Table 6 on page 31.

Horizontal Attributes

A horizontally defined table structure, as shown in Data Table 6 on page 31, is a result of "traditional" or literal data modeling. The Lesson Learned example is shown in Data Table 1 on page 17. If done well, the literal table structure produces a very straightforward and comprehensible database design.

Data Table 6: Horizontally Defined Table

Key	Attribute A	Attribute B	Attribute C	Attribute D	Attribute E
0	A1	B1	C3	D1	E1
1	A2	B3	C2	D2	E2

The drawback comes when new attributes need to be added to the table, as the columns are in a fixed structure. To add a new attribute, a new column must be added. This can be done by altering the table with a Data Definition Language (DDL) command. Altering the table is appropriate only if the new column allows null values, as the database will create null values for the new attribute for each existing row. If the new attribute does not allow null values for any row, then more drastic measures must be taken to preserve the data and recreate the table with the new non-nullable column.

This is not the end of the required adjustments, however. Any queries that retrieve data from the table will require modifications to their Structured Query Language (SQL) SELECT clauses. Data Manipulation Language (DML) commands like UPDATE and INSERT must be revised to account for the new column. Of course any related programs will likewise require adjustments. In most information systems this is not a problem, of course, if the data structure is stable.

Vertical Attributes

The same data depicted in Data Table 6 on page 31 is transformed into an inverted table, shown in Data Table 7 on page 32

Data Table 7: Vertically Defined or Inverted Table

Key	Attribute	DataValue
0	A	A1
0	B	B1
0	C	C1
0	D	D1
0	E	E1
1	A	A2
1	B	B2
1	C	C2
1	D	D2
1	E	E2

The advantage of this type table definition, and indeed its primary purpose, is to allow new attributes to be added when required without altering data structures and programs. This data structure is best employed when there is little or no *a priori* knowledge of the incoming data. Inverted tables are highly flexible. The drawback is that the attributes (fact types) are not readily obvious in the data structure. Because the table is inverted, individual attributes are not easily indexed for performance during retrievals. If indexed properly, however, the inverted table can essentially index every attribute. Database access and indexing will be discussed in the section entitled, *Indexing for Performance.*

Grammar to Control the Inverted Table

Left alone, as mentioned above, the inverted structure requires a great deal of knowledge about the actual data stored in the table to make use of the data. To provide meaning and to facilitate utilization of the recorded data, a set of Grammar or value control tables is employed. The term Grammar is used because these tables define the rules by which data may be stored in the inverted table.

Before explaining the Grammar tables, the distinction between object types and object instances needs to be discussed. The following example is used to illustrate this distinction:

> For an Employee (an object type), an Address, Social Security Number, Salary and a few other attributes are recorded. For a *specific* employee named John Doe (an object instance of the type Employee), his Social Security Number is recorded as 123-45-6789, and his

Salary as $50,000, etc. The object type is a description or grammar that will be applied to the actual data that will be captured for instances of a particular type.

The basic Grammar structure is composed of three primary tables, which are listed and described below.

Item (or Object) Type: This table simply delineates the possible entities. This is generally achieved with an identifying key and a unique name.

Data Descriptor: This table enumerates the list of possible Attribute Names (or Descriptors) that may be utilized by any of the Item Types. Again, this is generally done with an identifying key and a unique name. Depending on how the Attribute is to be used in the inverted table, data typing and utilization (i.e. mandatory, cardinality) can also be recorded for each Descriptor.

Item Type Descriptors: This table is a many-to-many junction between the lists of Item Types and Data Descriptors. Recorded in this table is the list describing the valid Data Descriptors (Attributes) for each Item Type that can be recorded for instances of that type. Additional information about the usage of the Data Descriptor for the Item Type is recorded including whether the Descriptor must be recorded for any Item instance and if more than one value may be recorded. This information can be used to dynamically build user interfaces or even batch data loading routines. To facilitate this dynamic property, a sequence number can be recorded which specifies the order that the Attributes will be recorded in the inverted table or are shown on a user interface screen.

The Grammar tables describe the structure of the Attributes that will be recorded for specific instances of Item Types, which we will call Items (see Model 9 on page 45). The Grammar can also specify controls that can be applied to Attribute values. Three types of value controls are identified:

- Freeform—there is no value control
- Range—data values must fall within a specified range
- PickList—data values must be selected from a predefined list of values

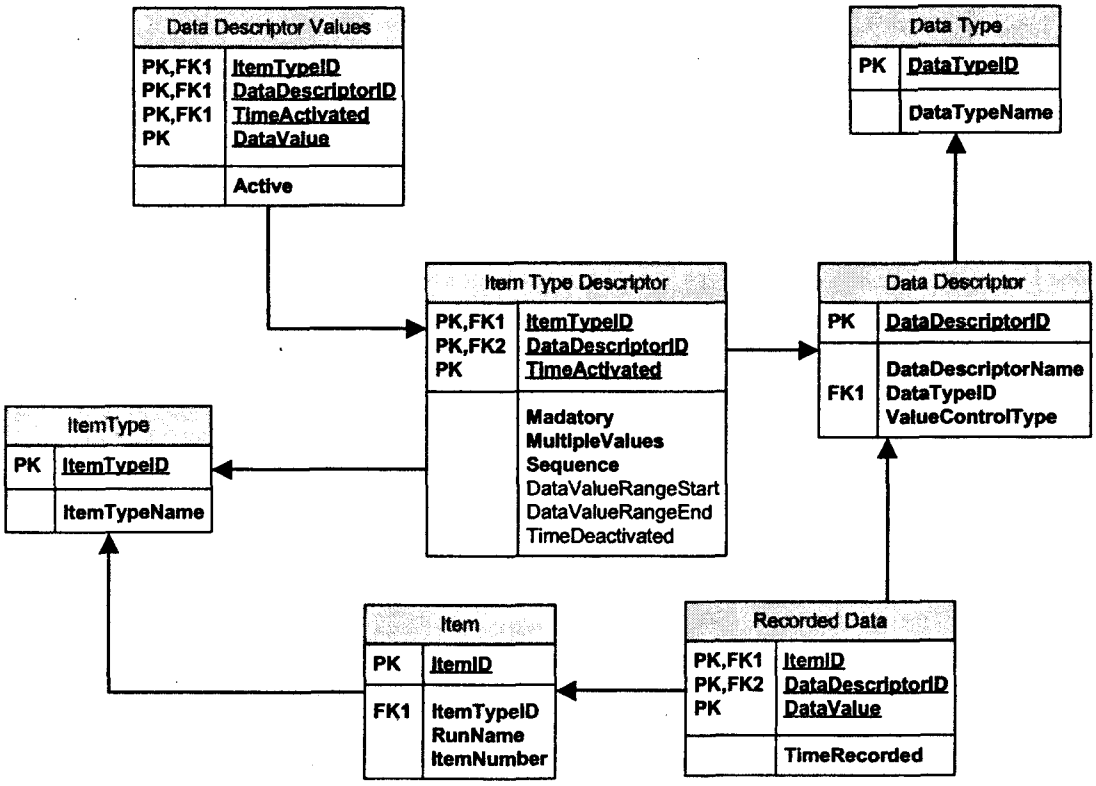
This Value Control Type attribute can be applied to either or both the Data Descriptor and Item Type Descriptor tables. If applied at the Data Descriptor level, the Attribute implies that any Item Type that uses that Descriptor will utilize the specified value control. If applied to the Item Type Descriptor, the implication is that each Item Type will specify its own unique value controls. If the value control is applied at both levels, the implication must be left to the employing application. The most obvious implication, however, is that the specification at the Data Descriptor level is a default and any specification at the Item Type Descriptor level is an override.

The three different types of Value Controls require some additional Grammar mechanisms. Clearly the Freeform value control requires no additional information. The Range control type requires two additional attributes: the range start and end values. The PickList value control requires a new table that enumerates the list of possible valid values.

Putting It All Together

In the original description of the inverted table, discussed in the section entitled *Implementation of the Generalized Data Structure*, the column "key" is shown. Its importance is obvious because it is the tie that binds the various Recorded Data Descriptors together to form a single record or instance. As previously stated, a specific instance of an Item Type is called an Item. Items are the ties that bind sets of entries together in the inverted table. Item is implemented as a table with a unique identifier for every instance. For each Item, the Item's Item Type and any identifying information for the Item are recorded. Data Table 8 on page 34 is an Entity Relationship diagram of the tables used for the Pu Disposition simulation project.

Data Table 8: Pu Disposition Simulation Project Relational Database Tables



Database Implications

The grammar-driven data capture relational data structure is really no different than any other relational database implementation. The same rules and analysis techniques still apply. However, there are database considerations for generalized data structures, as discussed below.

Storage Considerations

As opposed to other database implementations, the bulk of the data in a generalized structure is concentrated into two tables: the Item table and the Recorded Data table. There will be one entry in the Item table for every instance to be recorded. The Recorded Data table will contain one entry for each required attribute. An estimate of the expected optional attributes for a given Item Type should be made for entries in the Recorded Data table. That number should be augmented to include an estimate of any multi-valued attributes.

Indexing for Performance

The Item and the Recorded Data tables, in fact all the tables, should be indexed with a unique index for the primary keys. Many relational databases automatically uniquely index primary keys. Any attributes in the Item table that are used to identify specific instances should be indexed. In the Pu Disposition Simulation example, the ItemID is essentially a surrogate key for the ItemTypeID, RunName, and ItemNumber. A second unique index was created to enforce this relationship. Access is primarily to instances as groups given, an Item Type and a Simulation Run Name. Therefore, a clustering index is created on RunName, ItemTypeID, and ItemNumber, in that order. A clustering index was used to order the data physically in the table.

Most queries for instance data will in essence be a listing of instance Attributes for a given Item Type. This type of query is accomplished by joining the Item and Recorded Data tables and is facilitated by the secondary indices on the Item table and the primary key index of the Recorded Data table, assuming the primary key index of the Recorded Data table is created with the ItemID as the first column.

Queries that work the other direction—asking for a list of Items with a specific Attribute value—will be facilitated by a secondary index on Recorded Data where the columns DataDescriptorID and DataValue are specified first before ItemID.

Querying the Inverted Table

As mentioned earlier, querying the inverted table requires an understanding of the data housed in it. This knowledge is also housed in the Grammar, which would facilitate constructing the required SQL. Assuming such knowledge, devising queries to list information from the table becomes prescriptive.

Assume that an inverted table is used to store data for a single type of instance, and that it was created as an inverted table because it has a possible large number of null attributes per instance (this may or may not be reasonable from a database design perspective, but that is not the point here). Also assume that only attribute "A" is required and attributes "B" through "E" may not exist for any given instance. If it is desired to list all the instances in a tabular fashion with columns and rows allowing for null attributes, how is this achieved with Structure Query Language (SQL)? The following set of examples illustrate a solution:

The Data Definition Language (DDL) to create the example is:

```
CREATE TABLE InvertedTable (
        KeyCol      INT         NOT NULL,
        Attribute   CHAR(10)    NOT NULL,
        DataValue   CHAR(10)    NOT NULL)
```

For simplification of the SQL, a structure is needed that provides a one-to-many relationship between the key identifier and attributes. For this example, this will be achieved with the following view of the table:

```
CREATE VIEW Keys AS
        SELECT DISTINCT (KeyCol) KeyCol
        FROM InvertedTable .
```

The SQL needed to produce the desired listing follows:

```
SELECT      Keys. KeyCol

,           attrA.DataValue     A
,           attrB.DataValue     B
,           attrC.DataValue     C
,           attrD.DataValue     D
,           attrE.DataValue     E

FROM        Keys

,           InvertedTable attrA
,           InvertedTable attrB
,           InvertedTable attrC
,           InvertedTable attrD
,           InvertedTable attrE

WHERE       Keys.keyCol   =     attrA.KeyCol
AND         attrA.Attribute =   'A'
AND         Keys.keyCol   *=    attrB.KeyCol
AND         attrB.Attribute =   'B'
AND         Keys.keyCol   *=    attrC.KeyCol
AND         attrC.Attribute =   'C'
AND         Keys.keyCol   *=    attrD.KeyCol
AND         attrD.Attribute =   'D'
AND         Keys.keyCol   *=    attrE.KeyCol
AND         attrE.Attribute =   'E'
```

If InvertedTable contains the population shown in Data Table 9 on page 37, then executing the query will produce the listing as shown in Data Table 10 on page 37.

Data Table 9: Populated InvertedTable

KeyCol	Attribute	DataValue
0	A	A1
0	B	B1
0	C	C1
0	D	D1
0	E	E1
1	A	A2
1	B	B2
1	C	C2
1	D	D2
1	E	E2
2	A	A1
2	E	E1
3	A	A2
3	B	B2
4	A	A1
4	A	A2
4	B	B2
4	C	C1
4	D	D1
4	D	D2
4	E	E1

Data Table 10: Results of Query Execution

KeyCol	A	B	C	D	E
0	A1	B1	C1	D1	E1
1	A2	B2	C2	D2	E2
2	A1	NULL	NULL	NULL	E1
3	A2	B2	NULL	NULL	NULL
4	A1	B2	C1	D1	E1
4	A1	B2	C1	D2	E1
4	A2	B2	C1	D1	E1
4	A2	B2	C1	D2	E1

Standard SQL has the ability to alias any given table in a query's FROM clause. This query takes advantage of this capability to join the InvertedTable to itself, once for each attribute.

In general, a prescription for querying an inverted table structure can be defined. For each entity attribute that will be selected, aggregated, or filtered upon, include in the query:

- A FROM clause predicate for the inverted table with an alias.
- A WHERE clause predicate to join the inverted table to the entity table.
- If the listing is to include any optional attributes, then an "outer join" designator should be added to the join predicate.
- A WHERE clause predicate to designate the attribute.
- If any filtering is required on the attribute, a WHERE clause predicate or predicates to filter on the data value.
- Appropriate SELECT clause entries for either listing or aggregating of attributes data values.

It should be noted that if an Attribute which allows multiple values is to be recorded for any given object, the result set would be expanded accordingly. This is analogous to joining two tables where a one-to-many relationship exists. This should be kept in mind, especially when applying the aggregation functions SUM and AVG.

Conclusion

A generalized data model is the result of meta-level modeling of the verified and validated literal data model. The literal model's fact types are transformed into Grammar and Data Capture fact types. These modeling activities result in two or three data models: literal, subtyped, and generalized. These three models are semantically equivalent; however, the resulting tables are not. It is quite likely that the implemented data model will neither be the most literal or the most generalized version.

Although there is no simple formula to determine the optimal data model to implement, the exercise of generalizing the literal model provides the benefit of discovery of a potentially better solution and also helps in decision making. Transitioning understanding of the process from literal to generalized to the implementation team is highly beneficial. It provides the domain expert's point of view, so that no understanding gained in the analysis process is lost during implementation. It also provides a history of the thought process from literal to generalized, which engages the implementors in a wider scope of the software life cycle than just coding and testing. The result is a higher quality product and higher interest for the development team.

With all of the foregoing discussion of generalization, one is justified in asking the question, "At what point can you generalize too much?" Consider an application where there are many tables that are defined with a unique identifier, a name column, and pertinent additional attributes. An analyst could conclude that each table is a subtype of an abstracted entity called Named Object. This table would be defined with a unique identifier as the primary key and two attributes: SubType and Name. All the children tables would simply refer to this parent table for the name. While this may allow the children tables to be more concise, it concentrates a great deal of information pointlessly in a single table. Every query that joins two or more children tables will require a join to the Name table for each child table. Such a design is

more than cumbersome in terms of query design, database performance will be compromised because of the excessive joining and concentration of data in the Name table. In this example, the generalization (subtyping) took place simply by looking for commonality rather than increased functionality; and therefore should not be implemented.

The development team needs to weigh the advantages of a generalized model with the costs of implementation. What is the probability that the data will change? Will the rules or processes change? What are the associated risks with those changes? How easily can the development team handle meta-level models?

Advantages of generalization include the following:

- Allows for more in depth understanding of the problem, data, and data relationships
- Incorporates integrity constraints into the data structure (the "allowable" values)
- Provides flexibility and growth without changing the database structure
- Maps the customer-verified literal model to the generalized model for verification purposes
- Enables easier maintenance of the data structure
- Provides for reuse of portions of generalized data structures for other projects. The instances of the generalized data fact types will change, but not the structure

One cost to be considered with implementing a generalized data structure is that not all relationships and integrity constraints can be enforced at database level. Instead, this must be done at the process level, which requires additional code. This additional code is prone to errors and therefore more testing must be performed. However, if a literal model is implemented, integrity constraints on the data could change which would require modifications to the data structure and code.

The process of generalization absolutely creates a greater understanding of the problem and enables the designer to make a more intelligent decision about what design to implement. Technically, implementing the generalized model provides the most benefit if the data and process requirements have a high probability of changing; however, if you can't communicate understanding of the generalized model to the entire team, it may actually become a disadvantage. If the generalized model is communicated properly and understood by the team, the result can be the implementation of a high-quality, robust software product.

Bibliography

C. J. Date, *What Not How, The Business Rules Approach to Application Development.* Addison-Wesley, 2000.

Shelley M. Eaton, Building a Bridge with the Customer to Facilitate Collecting and Validating Information in Modeling Sessions, in *NIAM-ISDM 1994 Conference*, held in Albuquerque, New Mexico, August 24-26, 1994. Sandia National Laboratories, Albuquerque, New Mexico, 1994.

Terry Halpin, *Information Modeling and Relational Databases*. Morgan Kaufmann Publishers, San Francisco, California, 2001.

David C. Hay, *Data Model Patterns, Conventions of Thought*. Dorset House Publishing, New York, New York, 1996.

Michael Jackson, *Problem Frames, Analyzing and Structuring Software Development Problems.* ACM Press Books, Great Britain, 2001.

Gregor Kiczales, Jim des Rivieres, and Daniel G. Bobrow, *The Art of the Metaobject Protocol.* The MIT Press, Cambridge, Massachusetts, 1995.

James Martin and James J. Odell, *Object-Oriented Methods, A Foundation.* Prentice Hall, PTR, 1998.

G. M. Nijssen and T. A. Halpin, *Conceptual Schema and Relational Database Design.* Prentice Hall, of Australia, 1989.

Glossary

Data Capture. The fact types of the generalized data model used to store persistent data in a relational database.

Data Model. A model that represents the communicated reality of the data and their relationships and constraints for a particular area of interest or universe of discourse.

Data Modeler. A person who represents communicated information about data, data relationships, and data constraints from domain experts in a formalized syntax.

Data Object. An idea, notion, or concept that is applied to things or objects in our awareness, which is represented in data models. Objects can be related to other objects and are described in data models.

Domain. A specific area of interest or a bounded problem.

Domain Expert. A person who has a thorough understanding about the business or problem being addressed, along with the data and processes involved in that business or problem. This includes the data, the data relationships, how the data are used, and the constraints on the data.

Fact Type. A statement of truth specific to the area of interest that is a generalization of a communicated example set. A fact type is part of an ORM data model.

Grammar. The fact types of the generalized data model which control or allow certain values, constraints, or relationships to be captures in the relational database.

Horizontally Defined Table. A typical relational database table that represents instances of an entity. In these tables the entity is represented as a single entry in the table and the attributes are defined horizontally as columns.

Literal Data Model. The model which depicts the most specific data or the lowest level of generalization, also referred to as the conceptual model. This is the model that the domain expert verifies and validates.

Logical Model. A generalized or abstracted version of the literal model, which includes Grammar and Data Capture fact types.

Object Role Modeling (ORM). A method for modeling and querying an information system at the conceptual level, developed by Terry Halpin.

Physical Model. The detailed generalized or abstracted model that contains implementation fact types, data types, and data lengths. This is the model that is used to generate the DDL to create the data structure.

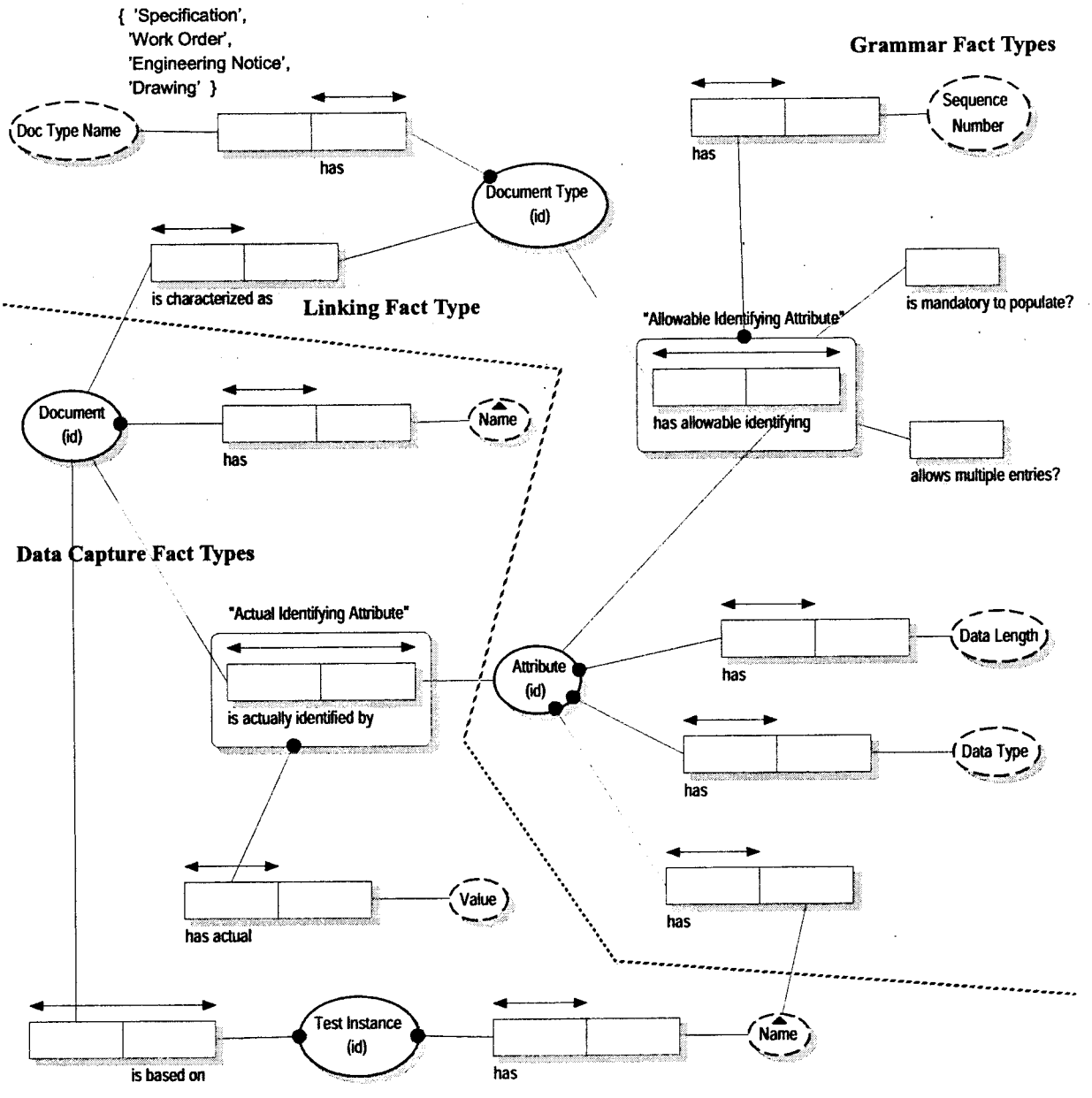
Vertically Defined Table. A relational database table where the entities are represented as multiple entries that are tied together by some key field and entry represents a single attribute of the entity.

Appendix A: Generalized Data Models and Resulting Relational Data Tables

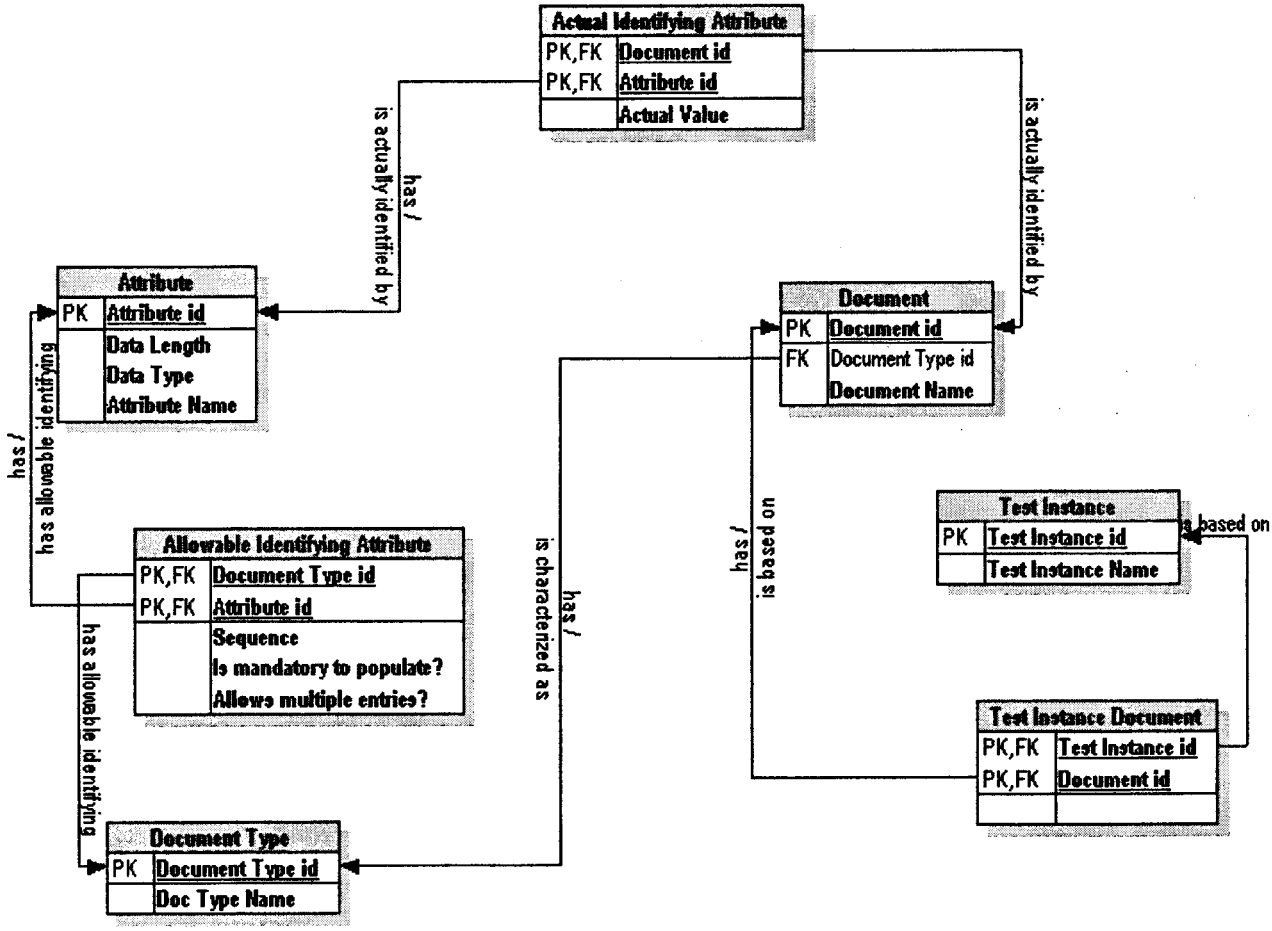
Model 8 on page 43 is another example of a generalized model, this one dealing with various kinds of documents. One of the data issues addressed by this model is that different kinds of documents have different kinds of unique identifiers and different attributes to describe them. Initially separate literal data models were created, each one handling one type of document. All of the literal models were then synthesized into a generalized data model, accommodating all known document types and unknown document types.

Model 9 on page 45 and Model 10 on page 47 are generalized models that can be used for any subject matter. "Thing" and "Attribute Type" replace specific domain labels. These models can be used as examples for study when creating other generalized models.

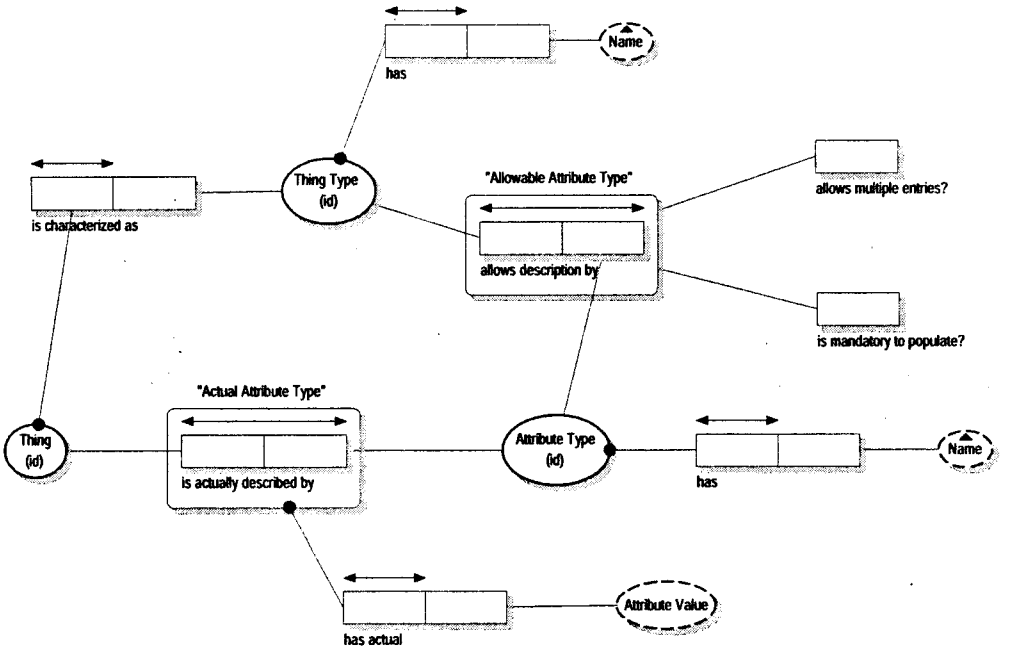
Model 8: Generalized Model for Documents and Document Attributes



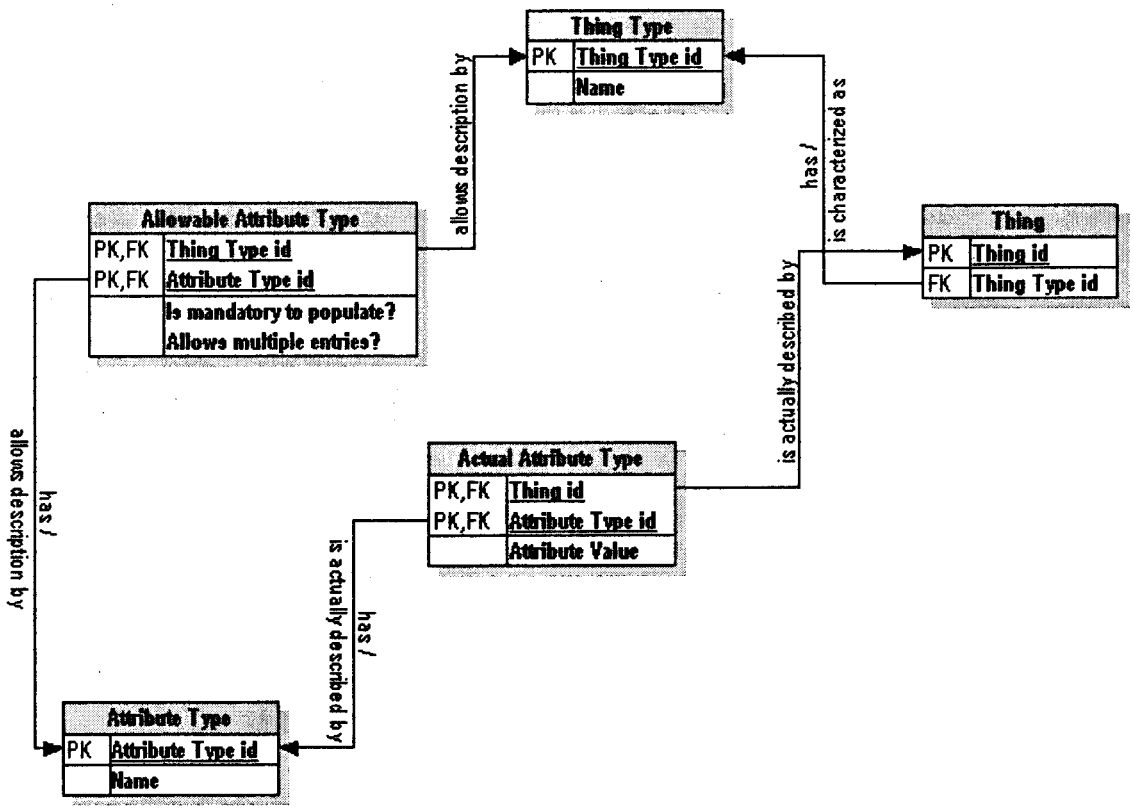
Data Table 11: Resulting Document Database Tables



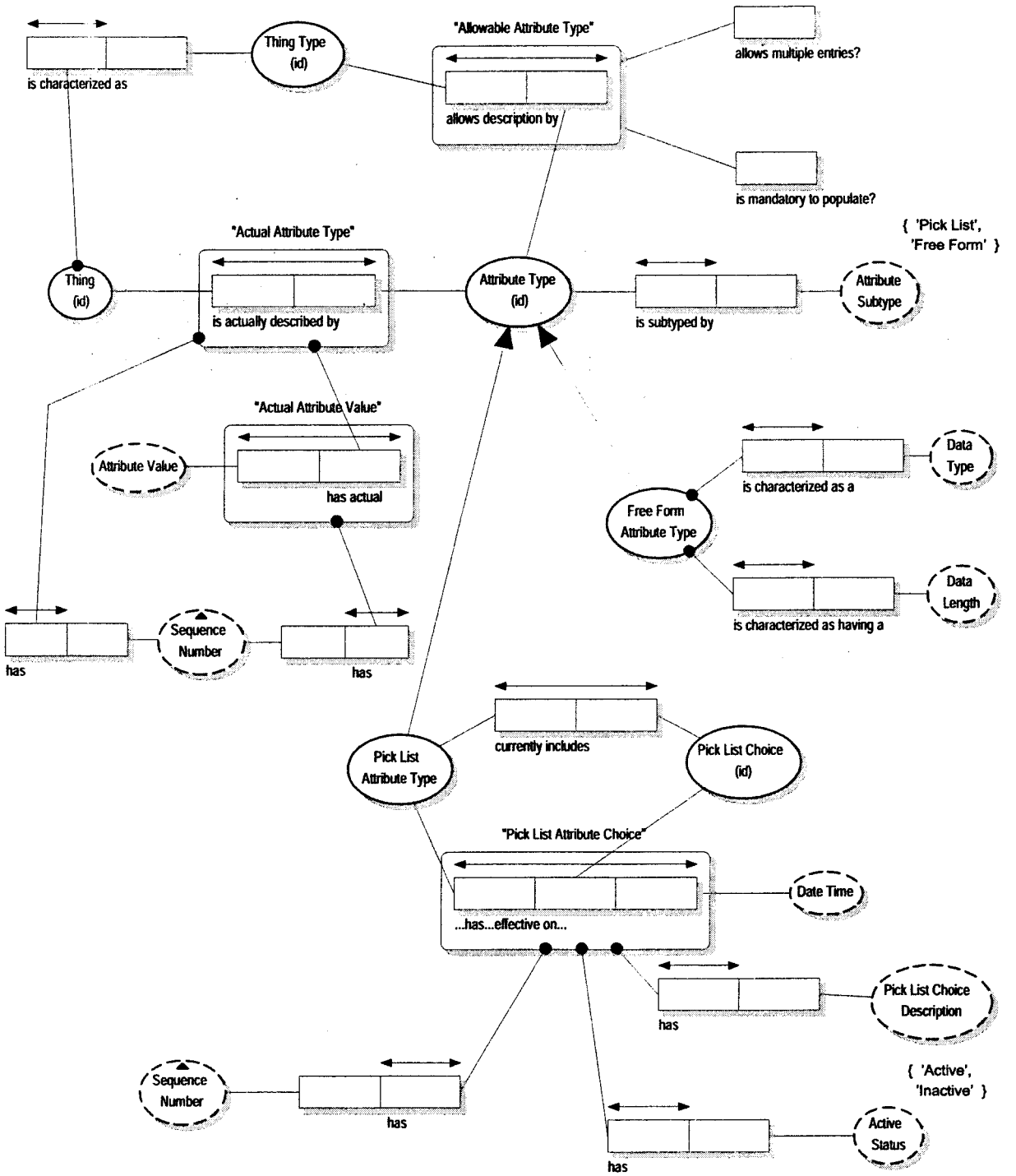
Model 9: Generalized Model for Things Described by Attributes



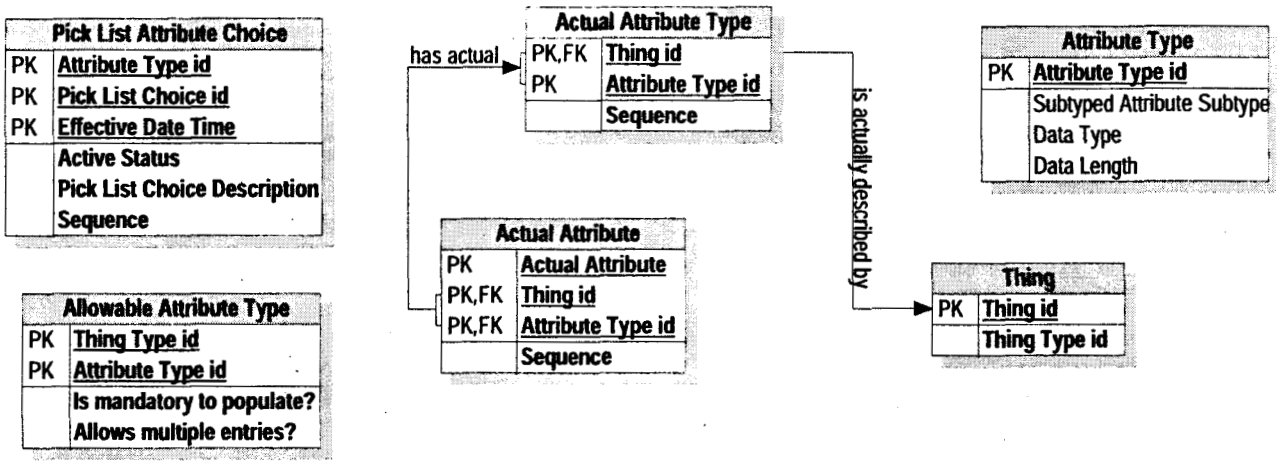
Data Table 12: Resulting Tables for Things Described by Attributes



Model 10: Expanded Generalized Model for Things and Their Attributes



Data Table 13: Resulting Relational Tables for the Expanded Generalized Model

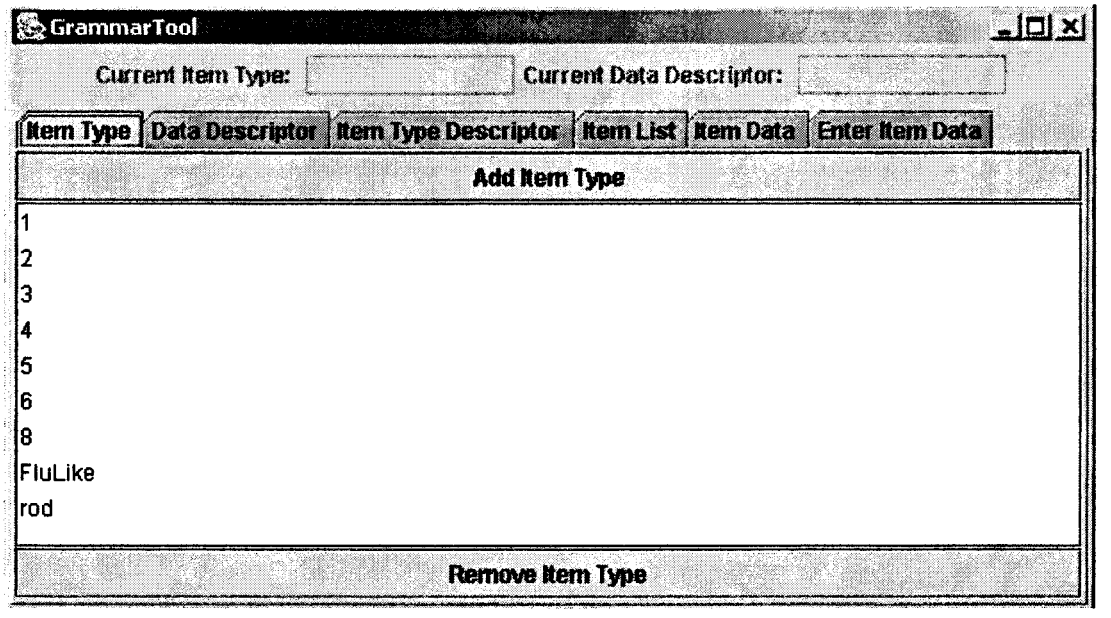


Appendix B: Generalized Data Structure in the Plutonium Disposition Application

A working prototype was built of a system to rapidly define and query simulation runs in the Plutonium Disposition project. There were two Java application tools in this system: the Grammar Tool and the Data Viewing Tool. The Grammar Tool allowed the development team to define the data that was needed to capture from each simulation run, and the Data Viewing Tool did just what its name implies.

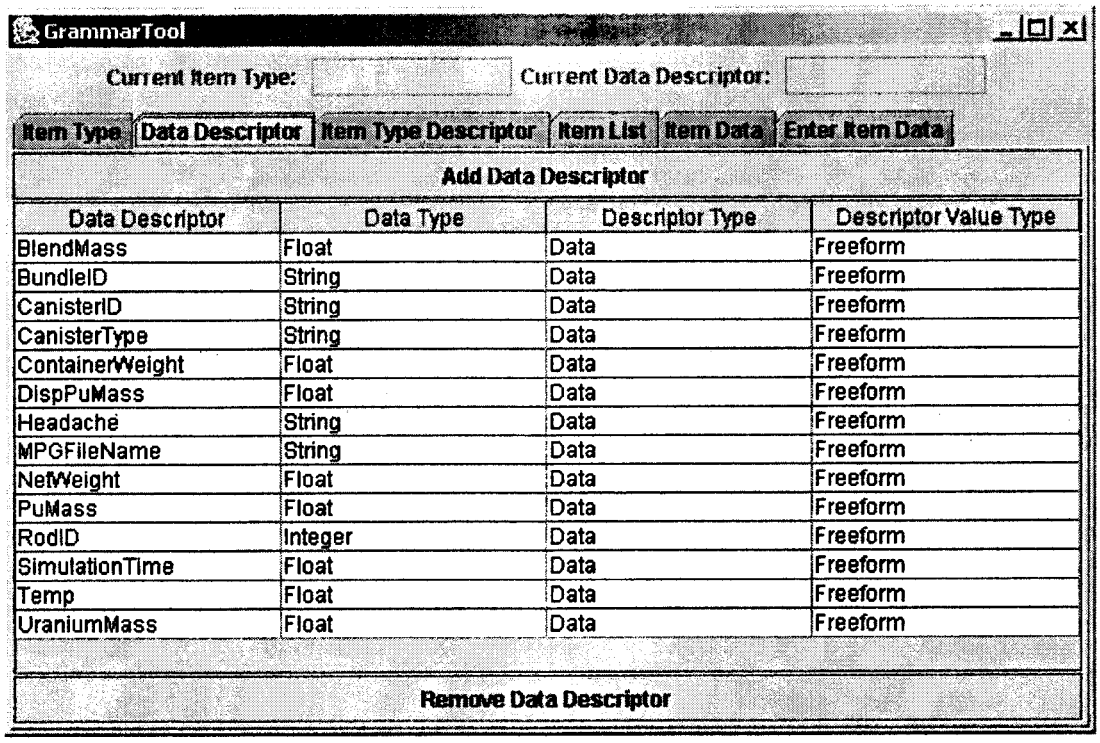
The following shows the Item Type screen of the Grammar Tool, which simply lists the entities or Items defined to the system. New Item Types can be added or deleted with this screen.

Screen 1: Item Type, Grammar Tool



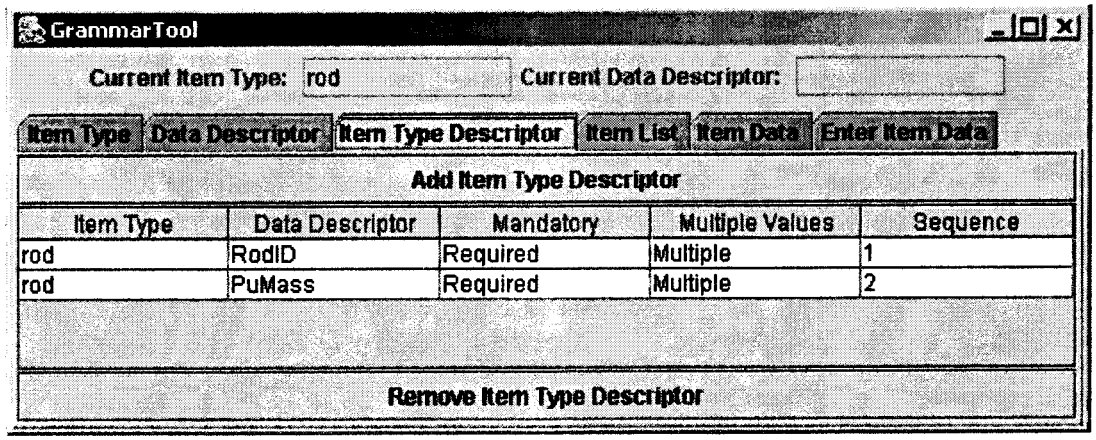
The following shows the Data Descriptor screen, which displays the possible attributes available to any Item Type. Data Descriptors can be added and deleted from this screen. Essentially, this screen defines the data format of the Data Descriptors.

Screen 2: Data Descriptor, Grammar Tool



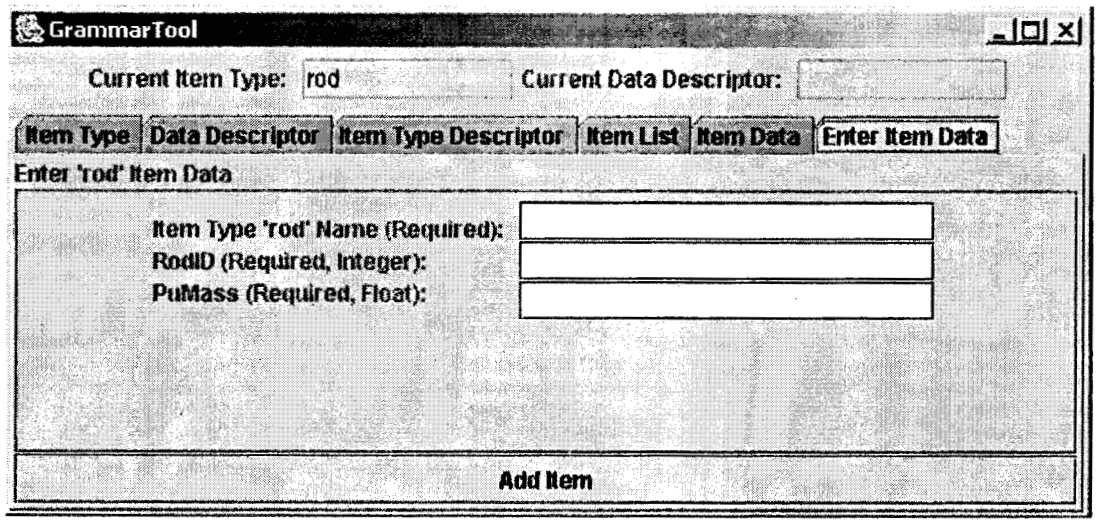
The following shows the Item Type Descriptor screen where the Data Descriptors are associated with the Item Types. Like the previous screens, the associations can be added or removed from this screen. At this point, the cardinality of the attributes is defined. The sequence is used to define screen layouts in later functions and for record layouts for data loading purposes.

Screen 3: Item Type Descriptor, Grammar Tool



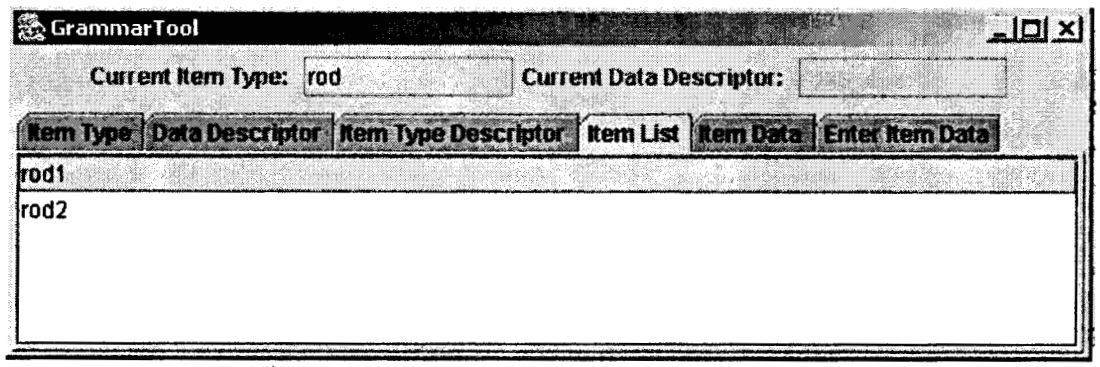
The following shows a Data Input screen, which was built from the grammar shown in the previous three figures. The Name field is a required field for all item instances and therefore was not listed in the user-defined grammar. The name is used for locating specific Item instances.

Screen 4: Data Input, Grammar Tool



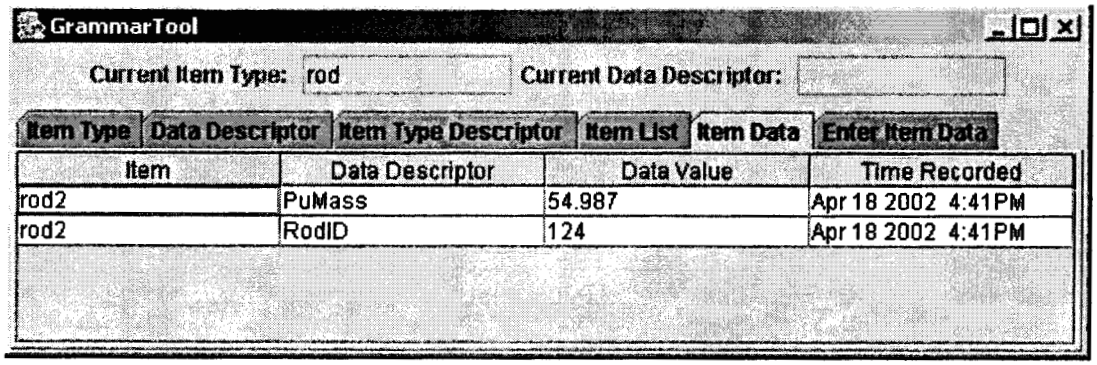
The following shows the Item List screen that simply lists the individual Item instances for a given Item Type.

Screen 5: Item List, Grammar Tool



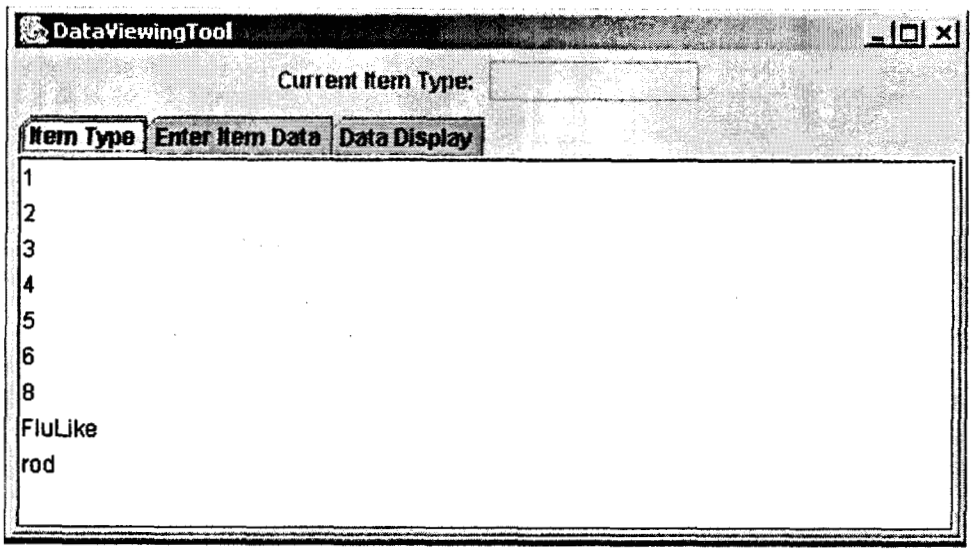
The following lists the specific Items data in inverted fashion directly from the recorded data table.

Screen 6: Inverted Items Listing, Grammar Tool



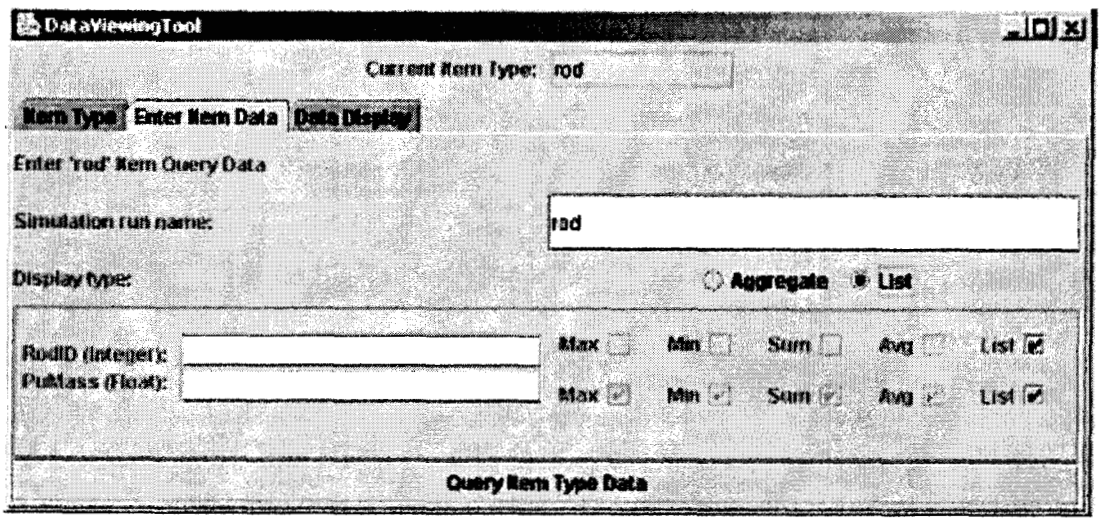
The Pu Disposition project's Data Viewing tool allows a user to view data for a given type and simulation run. The run names are a prefix of the item names. Like the Grammar Tool, the Data Viewing Tool starts with the user selecting an Item Type. The following shows this selection screen.

Screen 7: Item Type Selection, DataViewing Tool



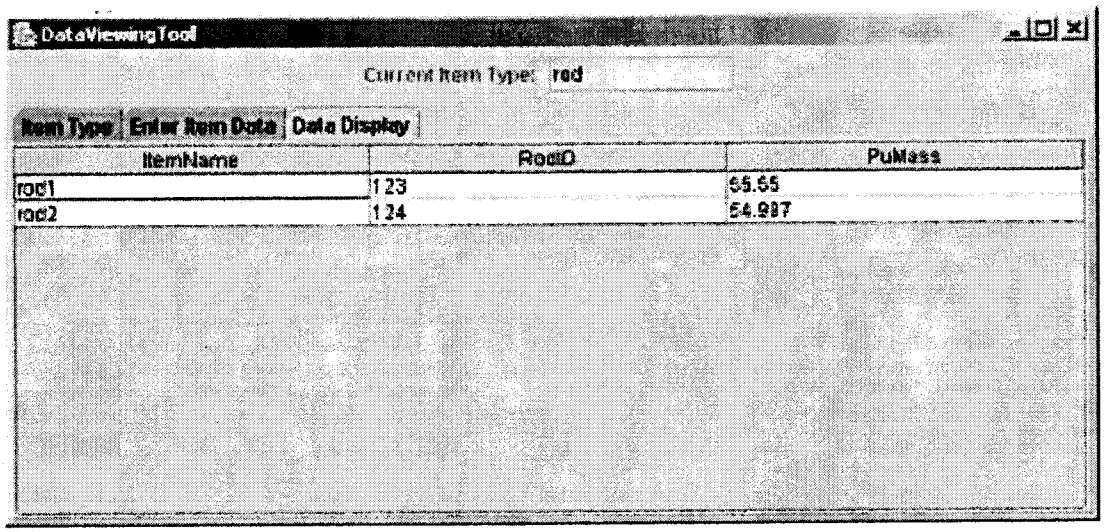
The following shows the Enter Item Data screen. With this screen a user specifies the simulation run name and the query parameters. If "List" is specified, then the tool simply displays the data for the item.

Screen 8: Enter Item Data, DataViewing Tool



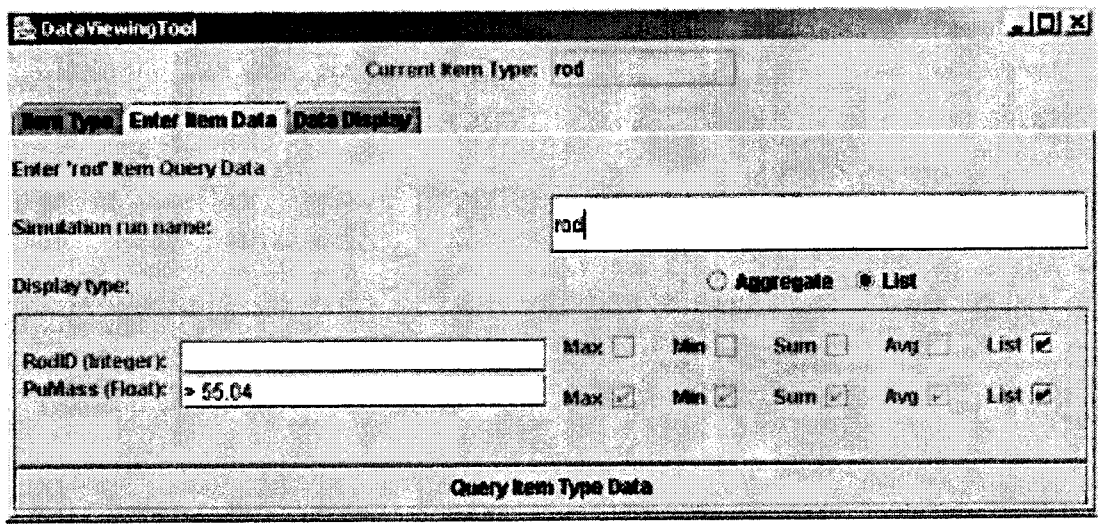
The following shows the Data Display screen as a result of pressing the Query Item Type Data button of the Enter Item Data screen.

Screen 9: Data Display, DataViewing Tool

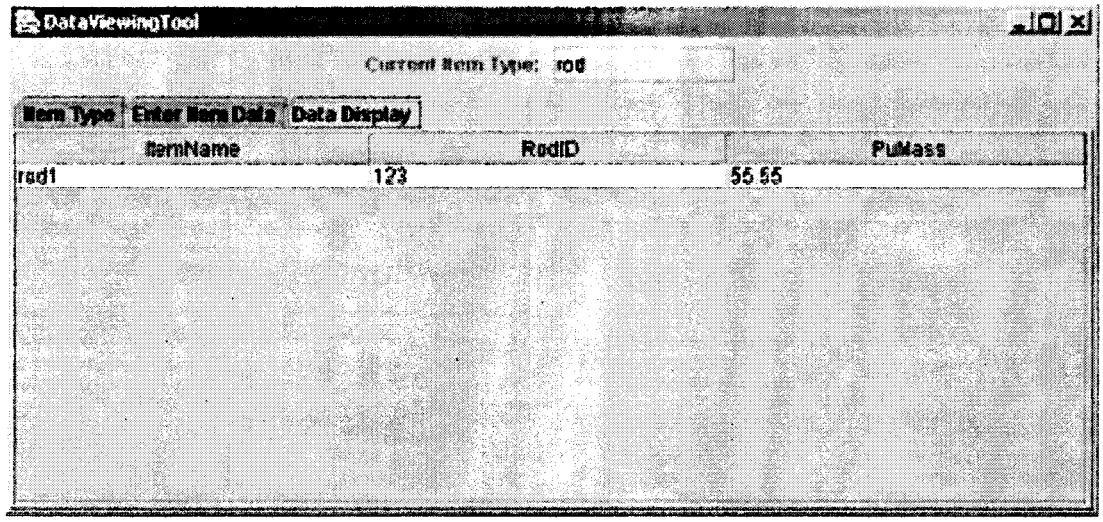


The following shows how filtering is specified in the Enter Item Data screen. Screen 11 on page 54 shows the result.

Screen 10: Enter Item Data, Data Viewing Tool

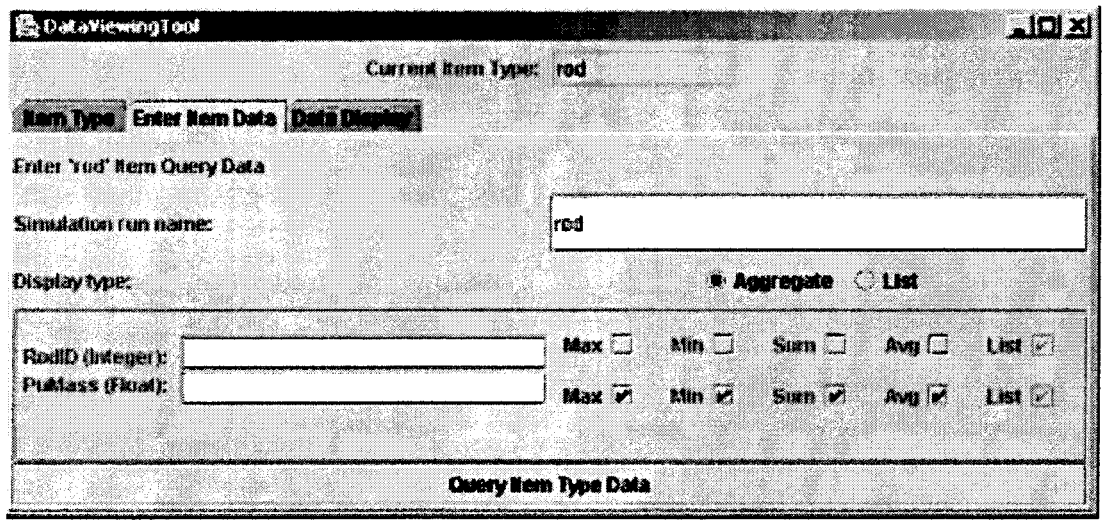


Screen 11: Results, Data Viewing Tool

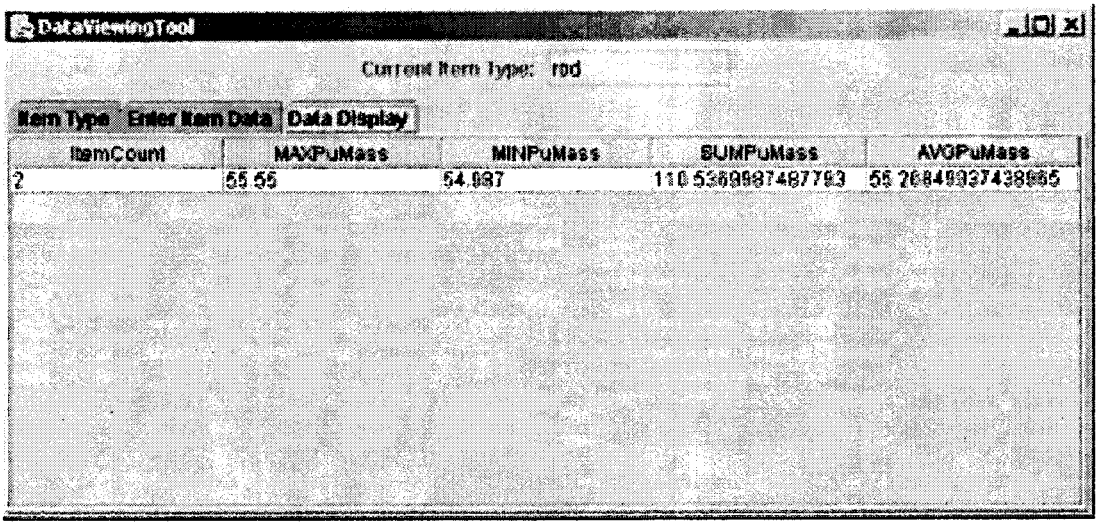


The following shows how aggregation functions are specified in the Enter Item Data screen. The following shows the result.

Screen 12: Enter Item Data for Aggregation Functions, Data Viewing Tool



Screen 13: Results of Aggregation Functions, Data Viewing Tool



Appendix C: ORM Graphical Notation Legend

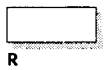
Objects

Entity type "A"

"A" identified by means of 1:1 reference mode "ref"

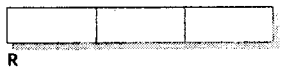
Value type "B" (strings or numbers)

Roles

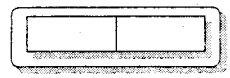


Unary predicate or role "R" (one role)

Binary predicate or role "R" (two roles)

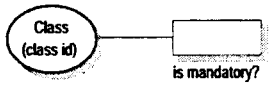


Ternary predicate or role "R" (three

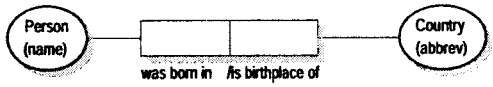


Nested object type

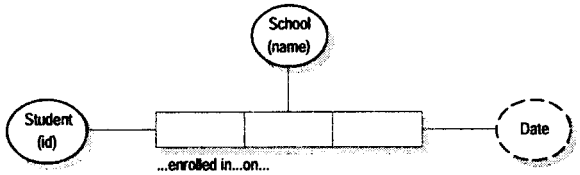
Fact Type Arity



Unary Fact Type

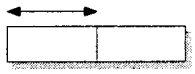


Binary Fact Type

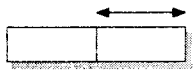


Ternary Fact Type

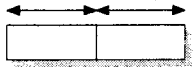
Uniqueness Constraints



No duplicate data are allowed in the first column (M:1 relationship)



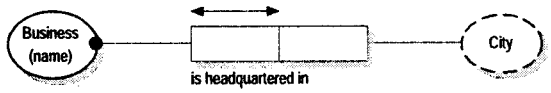
No duplicate data are allowed in the second column (1:M relationship)



Both the foregoing constraints apply (1:1 relationship)

No duplicate data are allowed in the combination of both columns (M:M relationship)

Mandatory Constraints



The role "is headquartered in" is mandatory for the population of the "Business" entity. In other words, if a Business Name is recorded, the City its headquartered in must also be recorded

Distribution:

1	0451	J. L. Mitchiner, 6541
1	0576	B. N. Malm, 6531
1	0660	E. L. Abeyta, 9329
1	0660	D. S. Cuyler, 9514
1	0660	C. D. Harrison, 9515
1	0660	E. M. Kephart, 9519
4	0660	A. J. Schrouf, 9515
1	0660	D. VanWesterinen, 9329
1	0661	M. K. Bencoe, 9519
1	0661	T. L. Carson, 9514
1	0661	G. E. Rivord, 9510
1	0775	M. J. Eaton, 5852
1	0775	J. W. Hatley, 5852
1	0805	C. M. Huber, 9523
1	0805	W. R. Mertens, 9523
1	0805	W. D. Swartz, 9329
1	0902	R. S. Joyce, 9514
1	0974	M. M. Hess, 6523
1	1137	O. H. Bray, 6544
5	1137	G. N. Conrad, 6544
4	1137	S. M. DeLand, 6544
10	1137	S. M. Eaton, 6536
1	1137	G. K. Froehlich, 6536
1	1137	K. L. Hiebert-Dodd, 6545
1	1137	A. L. Hodges, 6536
1	1137	S. L. Humphreys, 6544
1	1137	C. L. Jenkin, 6544
1	1137	M. T. McCornack, 6536
1	1137	M. J. Procopio, 6544
1	1137	W. A. Stubblefield, 6544
1	1137	M. A. Tebo, 6536
1	1137	M. M. Warrant, 6544
1	1137	P. Zhang, 6544
1	1138	D. P. Gallegos, 6533
1	1138	P. L. Vaughan, 6533
1	1138	E. R. Young, 6532
1	9018	Central Technical Files, 8945-1
2	0899	Technical Library, 9616

1	0612	Review & Approval Desk, 9612 For DOE/OSTI