Automatic Software Diagram Layout Within Metaview

A Thesis Submitted to the College of

Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the Degree of

Master of Science

in the

Department of Computational Science

University of Saskatchewan

Saskatoon, Saskatchewan

by

Robert Donn Morton

June, 1995

PERMISSION TO USE

ABSTRACT

Metaview is a meta-CASE tool used in the development of software systems. The development of software systems requires the creation of software diagrams. Metaview currently allows developers to generate diagrams manually with the aid of a graphical editor tool, MGED (Metaview Graphical EDitor). There is a clear need to have some capability of automatically generating software diagrams within the Metaview system.

This thesis explores and evaluates a method which will allow a software developer to take a specification database within Metaview and automatically lay it out according to specific criteria governed by a *placement algebra*. The method explored is manifested in the Automatic Diagram Layout system, abbreviated ADL, and it is implemented as a prototype tool within the Metaview system.

The ADL System is composed of three components. These components are the *Diagram Layout Language (DLL) compiler*, *ADL Tool*, and the *ADL Edge Router*. A methods engineer encodes the placement algebra within a DLL program for a given diagram type (or method). The DLL compiler translates the DLL source into modeling tables. The ADL Tool then uses these tables to generate layout constraint code for the corresponding method's specification databases. The generated layout constraint code is then input to the Incremental Constraint-based Object Layout Algortithm (ICOLA) platform. ICOLA uses this code to generate the specific diagram. The generated diagram is a graphical visualization of the textual specification database.

Currently, the ADL system has generated two types of diagrams. These diagram types are the Data Flow and the Entity Relationship diagrams used primarily in the structured analysis and design paradigm.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Dr. J.P. Tremblay, for the guidance, time and support that he has given me while I was preparing this thesis. I would like to thank Beth Millar for the many discussions that we had to keep me on track. I would also like to thank Dr. Anthony Kusalik, Dr. Eric Neufeld, and Dr. J. Szmigielski (external examiner from the Department of Mathematics and Statistics) for serving on my thesis committee. I would like to thank Greg Oster for the many hours that he put in to make ICOLA a more useable platform from which to base this research.

I would also like to thank my fiancée, Marylee, my family, and my friends for their support and encouragement.

Financial assistance was gratefully received from the Natural Sciences and Engineering Research Council, The University of Saskatchewan, my supervisor, Dr. J.P. Tremblay, and, my step father, Ed Bozak.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

Introduction

1.1 Motivation

Software diagrams are used for a variety of purposes [1, 25]:

- as a development tool to model various phases of the software life cycle; for example, Entity Relationship diagrams are used to model conceptual data design and Data Flow diagrams are used to model structured systems analysis and representations of requirements;
- as a documentation tool to represent aspects of the final product.

Despite the importance of diagrams in Software Engineering, diagrams are usually produced by hand or by means of a graphical editor. Graphical editors, however, provide little or no support for laying out the diagram. The responsibility of placing icons (polygon regions), and connections (arcs or edges) between icons, is thus left to the developer. The aforementioned tasks are referred to as the placement problem and the routing problem, respectively. Essentially, the problem of diagram layout, manual or otherwise, reduces to the these two subproblems. The placement problem refers to the determination of the absolute and relative locations of icons on a diagram's two-dimensional plane. The routing problem refers to the determination of a path for an edge connecting icons already placed on the diagram.

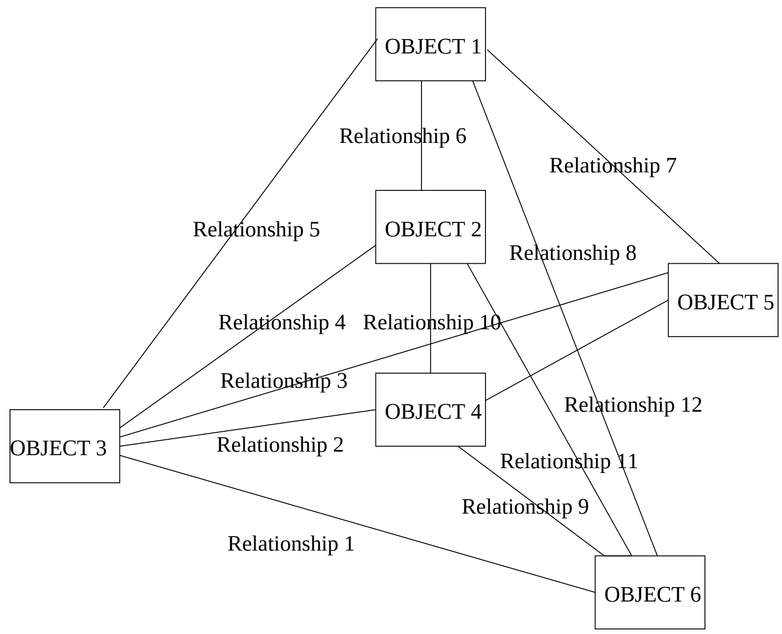


Figure 1.1: Diagram Produced by a Graphical Editor

The overall effect of manually generating diagrams is to either take vast amounts of time to generate legible and useful diagrams, or to throw together diagrams that have the legibility and usefulness commensurate with the time that it took to lay them out. The diagrams produced in the latter case can thus have a "spaghetti bowl" [1] style (See Figure 1.1).

Present technology does not provide adequate means for generating useful diagrams. One way of alleviating the problem of generating useful diagrams in a reasonable amount of time is to develop tools that automatically perform the layout of diagrams. Such a tool would generate a reasonable first draft of a diagram that a system developer would then improve to make the diagram more readable. The main advantages of this approach are [1]:

- reduction of production and maintenance costs;
- standardization of the graphic project documentation;
- increase of communication between developer and end user, and developers on the same project, by always having diagrams which are more likely to be up-to-date;

In order to develop such tools, we need to understand layout criteria usually adopted by human beings and find algorithms which are able to satisfy these criteria, at least in an approximate way. This task is not trivial.

Metaview [23] is a meta-CASE system (Chapter 2) in which software development paradigms are modeled. The paradigms that Metaview models correspond to software environments. Software environments usually have corresponding software development diagrams to provide a graphical means for the software developers to illustrate the conceptual meaning of their specifications. Currently, Metaview has no means to automatically generate software diagrams. There exists, however, a tool within Metaview called MGED, Metaview Graphical EDitor, which allows the developer to manually lay out a diagram based on a chosen conceptual and graphical environment. This is currently the only alternative that a system developer has available to develop a tool within the framework of Metaview. Future work in Metaview will allow other means to develop a software system. With this in mind, the problem of generating the corresponding diagrams for these software systems in some reasonable manner, other than manually, becomes apparent. What does one do to generate these diagrams?

Thus, a method which will allow a system developer to take a Metaview-defined software specification and then lay out the corresponding diagram automatically, needs to be developed. A possible approach to accomplish this task is as follows. A system which automatically lays out diagrams has been designed. This system is called ADL, which is an acronym for Automatic Diagram Layout. An architectural diagram, which includes the ADL system, appears in Figure 1.2. ADL, which will be considered to be a Metaview tool, has three main components. These components are shown in Figure 1.3, which is a decomposition of the ADL system of Figure 1.2.

The important component of ADL, a language DLL (Diagram Layout Language) [5], has been designed which allows a developer to refer to a Metaview software method. A methods engineer, using the language constructs, can describe the graphical specifics of the diagram type by specifying constraints on the Metaview object types. This reference to a Metaview software method is one of ADL's interfaces to the Metaview system. Essentially, the compiler for this language will take the diagram de-

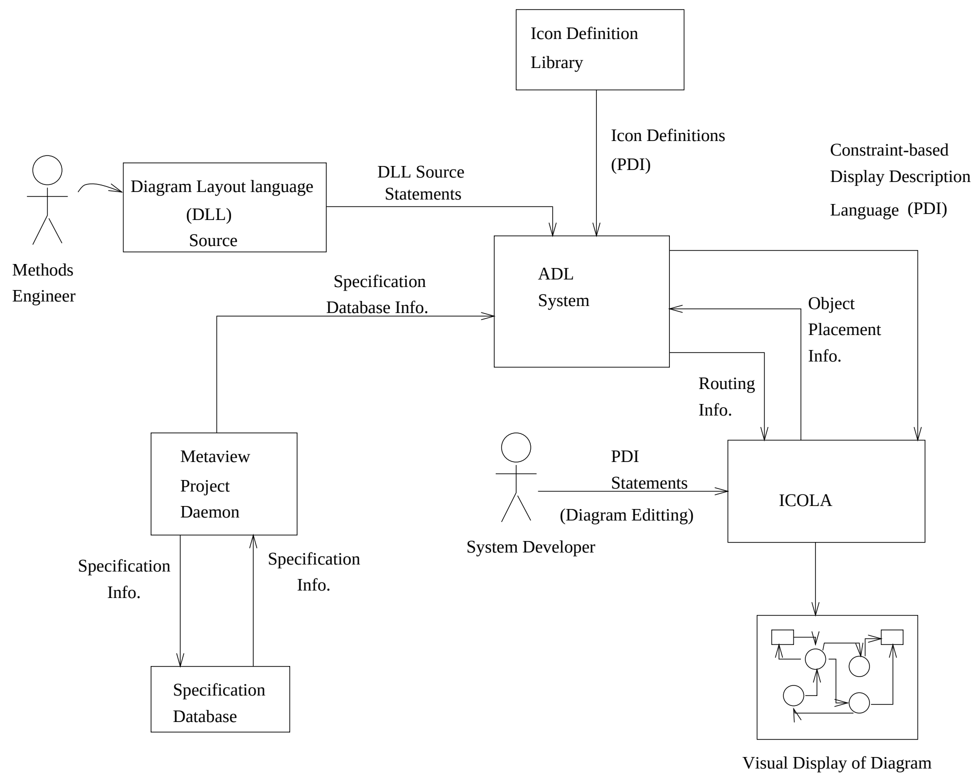


Figure 1.2: Automatic Diagram Layout System Block Diagram

scription source code, and generate diagram modeling tables which will be used later by the ADL tool as shown in Figure 1.3. The DLL compiler is invoked only once for each method, and is therefore detached from the automatic diagram layout process for a given method after that point. The ADL tool takes the diagram modeling tables, the software specification database information from Metaview, and a description of the icons to be used from an icon definition library and translates these to an object language which is the input language to an object layout platform. The ADL tool's use of Metaview specification information is another interface to the Metaview system. This object layout platform is external to the ADL system but plays an important role in diagram layout. The tentative layout platform for this endeavor is a system called ICOLA [15]. ICOLA stands for Incremental Constraint-based Object Layout Algorithm. This tool accepts a constraint-based display description (essentially the object code from the ADL tool) and generates a visual display of the desired diagram. Of course, there is no guarantee that the diagram that is generated will be exactly

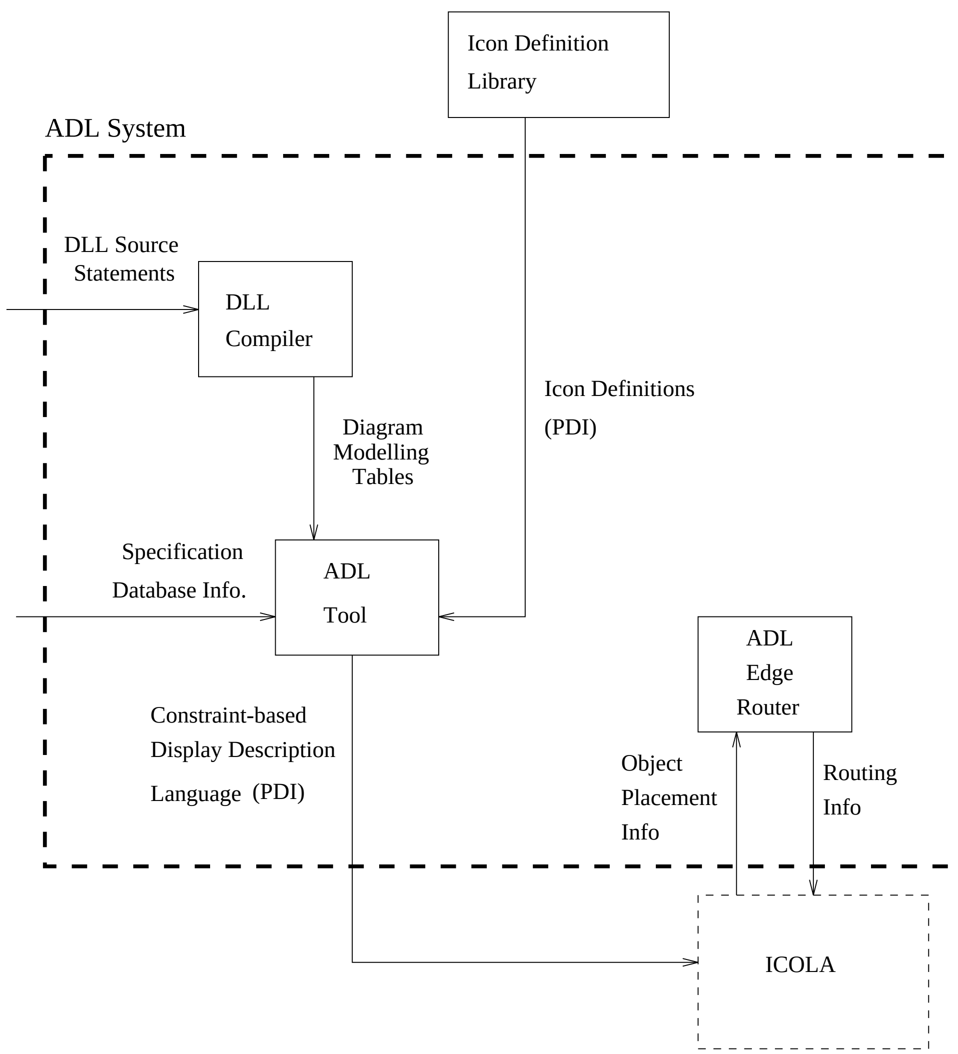


Figure 1.3: Decomposition of Automatic Diagram Layout System

what the developer intended. Thus, the ICOLA platform allows basic modifications to be made to the generated diagram. Future work will involve a more graphic-oriented, user-friendly means of making changes to the diagram.

This research is innovative in the sense that it is meant to be a significant first step in the process of solving the general diagram layout problem. The implemented prototype could be used as a starting point for further research, which will bring the software development community closer to having a useful generic software diagram layout tool. The current literature on this subject covers many automatic diagram layout systems but few researchers have actually attempted to lay out diagrams via a placement algebra. The placement algebra presented in this thesis is similar to that referred to in Protsko [20]. Essentially, the placement algebra is a model which concerns itself only with relative placement of objects on a two-dimensional plane. The research in this thesis explores this approach.

The main goal of this thesis is to explore the approach of automatic diagram layout which isolates the problems of placement and routing. This exploration involved implementing a prototype in which the placement problem is solved via placement algebra (to be discussed in detail later) and the routing problem is solved via a modified artificial intelligence grid search algorithm. The exploration has revealed many interesting aspects of diagram layout, some which were solved and others which were left as future work. The prototype discussed in this thesis is the essence of the exploration. It is not meant to be a commercially viable tool. It is meant as a research tool to take our understanding of the problems and pitfalls of generic automatic diagram layout further. The main goal has been accomplished.

1.2 Overview of Thesis

Chapter 2 examines the Metaview system in some detail. A discussion of the three levels of specification as well as a detailed description of its underlying data model is given.

Chapter 3 presents related work in the area of automatic diagram layout. In particular, diagram-specific and graph-theoretic approaches will be discussed along with their corresponding implementations.

Chapter 4 gives an overview of how ADL accomplishes the task of laying out diagrams. The chapter will discuss, in detail, the design of the entire system. It will describe how all of the components of the system fit together and how it relates to Metaview. This chapter also describes, in detail, the placement component of the ADL system. The design and implementation of this component is the essence of the research on which this thesis is based. A placement algebra was incorporated into the design of the ADL system via the Diagram Layout Language (DLL) grammar. This chapter will also discuss the DLL language from a design viewpoint. The ADL Tool and edge router are also discussed here at a high level.

Chapter 5 describes the compiler for DLL. The implementational details of the scanner, parser, symbol table, diagram modelling tables, and the action routines used to generate them, will be presented. The ADL Tool's main components will be decomposed and explained. This chapter also provides a description of the routing algorithm employed in the ADL system.

Chapter 6 illustrates the use of ADL, providing several diagrammatic examples within two methodologies. Two specification methodologies, which are currently supported within Metaview, include:

- Structured Systems Analysis (Data Flow Diagrams)
- Entity-Relationship Diagrams

A few diagrams based on the above methodologies will be provided and discussed in this chapter. Chapter 6 is essentially a results chapter. It evaluates the diagrams produced by ADL and the approach used in developing ADL.

Chapter 7 restates the motivation behind the ADL system and discusses the importance of producing diagrams automatically. The intended goals of the ADL project are reviewed and a discussion of how these goals were achieved is provided. It also states some conclusions about the research performed as well as future work in this area.

CHAPTER 2

Metaview

The ADL system is a component of the Metaview metasystem. In this chapter Metaview is described in enough detail to allow the reader to understand its use and how it can be enhanced by an automatic diagram generation tool.

The goal of a metasystem is to facilitate the construction of Computer Aided Software Engineering (CASE) environments. These environments are made up of the tools which developers use to build large software systems. The exact nature of the tools required depends largely on the methodology being employed, and it is advantageous to use a CASE environment whose toolset is well suited to the particular development task which is being carried out. However, custom-building a CASE environment for an individual software project is generally not feasible due to the great effort that would be required. The Metaview metasystem [23] is used to generate software specification environments for a wide class of specifications methodologies.

2.1 Metaview Architecture

Metaview is a metasystem approach to development of software support environments analogous to the compiler-compiler approach for defining and generating a compiler. This approach to automatic generation of software specification environments uses tables derived from formal definitions of a desired support environment. Figure 2.1 shows the architecture of the entire Metaview system. There are three levels of specification: the *meta* level, the *environment* level and the *user* level. The user, in this case, is the system developer.

At the meta level, the *meta definer* defines a *meta model* which should be sufficiently expressive to support the description of a large class of development environments. The meta model of Metaview is the EARA (Entity-Aggregate-Relationship-Attribute) model defined by McAllister [18]. This model is an extended entity-relationship data model and is described below. The meta definer should also oversee the activities of creating the database engine, the environment generator and the tool set facilities. The *tool definer* is responsible for creating the generic tools. Routines and definitions that define the functionality of the tool must be specified. This is accomplished by specializing the routines in the Tool Components Library to the functionality of the particular tool. These routines and definitions are combined with those in the Tool Components Library to create a Metaview Tools Library.

At the environment level, the *method definer* processes specification environment definitions, using the environment editors and compilers provided at the meta level. A method definition includes a formal description of the specification environment, as well as tool-specific information which is used to configure the individual generic tools. The specification environment is described formally as an environment model definition in the language EDL (Environment Definition Language). The EDL compiler processes the environment description, producing a set of environment tables. The environment tables and tool-specific information for each defined environment are stored in the method library. There also exists an Environment Constraint Language (ECL) which the method definer uses to enforce conceptual constraints on the instantiation of an environment/method by the software developer. The method definer encodes constraints in ECL and then compiles these constraints into the same form as the environment is encoded in. These compiled constraints are invoked whenever the software developer instantiates a method. The constraints ensure the integrity of the created specification database. The *software process engineer* is responsible for selecting and configuring the appropriate tools from the Metaview tools library.

At the user level, the support software provided by Metaview is used to help in

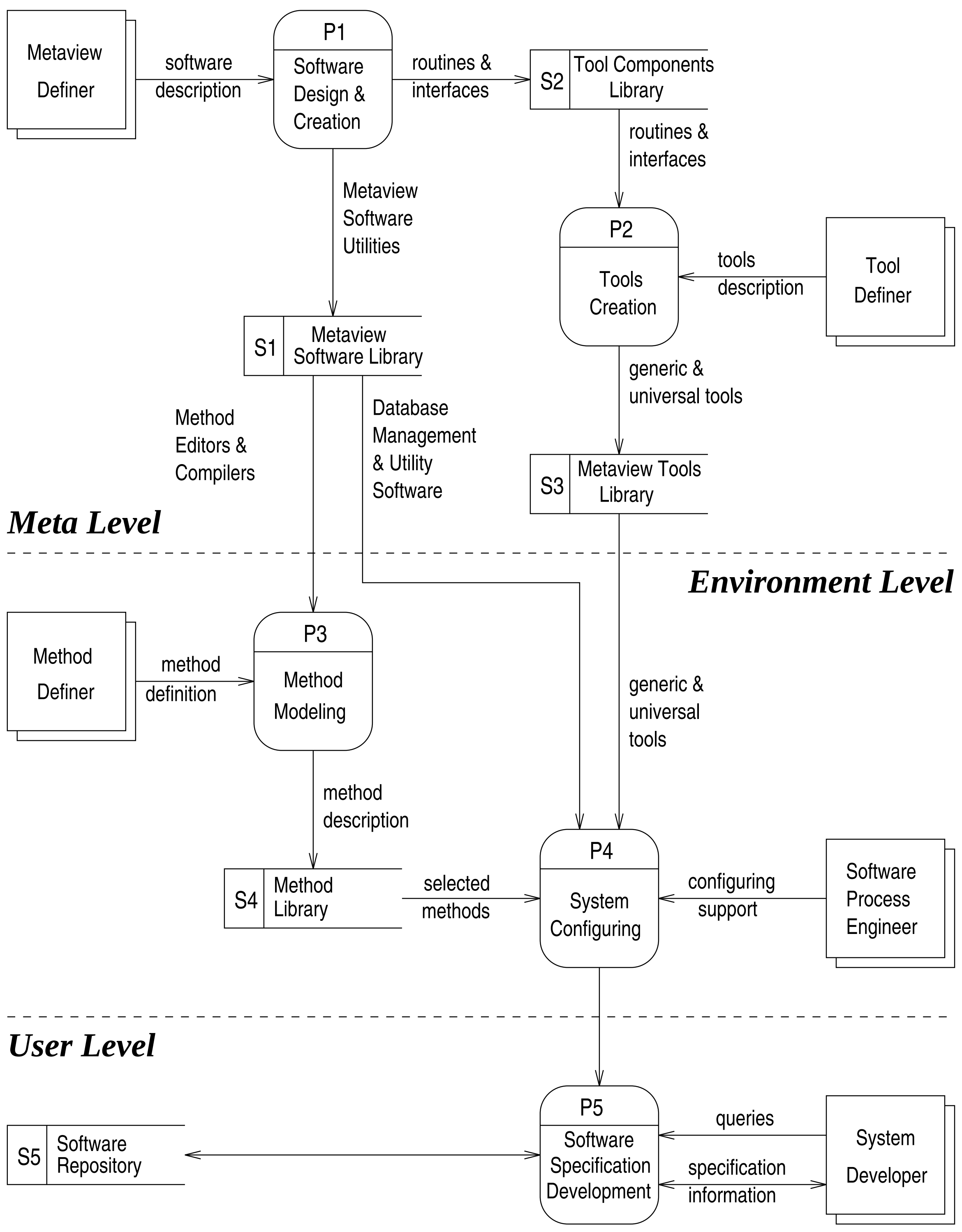


Figure 2.1: Metaview Architecture

the storage and analysis of a system specification. The user, who may be a system developer, analyst, or software requirements engineer, selects a particular specification environment (method) from those provided in the method library. The specification environment includes a set of configured database engine routines and specification support tools which assist the developer's interaction with the specification database. These facilities are used to store and analyze requirement specifications for any information processing system the analyst wants to define. Examples of such facilities are a textual input processor, an interactive graphical input processor, an automatic CASE diagram layout generator such as the one described in this thesis, and a specification explainer. The system developer uses these facilities to store and analyze specifications of the software system being defined. Each specification is stored in a specification database. This level allows a software developer to choose a software environment and instantiate it with a specification; for example, a data flow diagram based on a DFD environment. This instantiation of an environment is the specification database.

2.2 EARA Data Model

Metaview can provide support for a particular CASE methodology only if the necessary environment can be accurately specified in EDL. For this reason EDL must reflect a very expressive data model. The EARA model is based on the Entity-Relationship model of Chen[4] and contains features that permit advanced semantic-based data modeling beyond what the basic ER model provides. These features include the abstraction mechanism of aggregation, generalization, and classification. These advanced features are needed to adequately model the hierarchical nature of the relationships among components in a complex system.

The basic concepts provided by the EARA model are *entities*, *aggregates*, *relationships*, and *attributes*. *Entities* represent classes of real-world objects. Typical entities used in software environments are modules, procedures, documents, data, events, states, processes,variables, actions, data stores, etc.

Relationships are associations that exist between entities and/or aggregates. For example, tha fact that one module calls another can be expressed as a relationship between these modules. Similarly, we can use a relationship between a document and a module to indicate that the document describes the module. The entities and aggregates that take part in a relationship are called *participants* of that relationship. Relationships have *roles*. We assume that each relationship can be defined as a mapping from its roles to its participants. The roles determine the functions played by the participating entities.

To provide suitable support for complex objects, the model is capable of representing a heterogeneous collection of entities and relationships as an *aggregate*. The entities and relationships belonging to an aggregate are called *components* of that aggregate.

Very often the objects existing in a software environment do not have a simple, atomic structure, but rather represent complex subsystems. The subsystems can be represented by aggregates. To give the capability of representing hierarchical systems, or presenting a software specification at a desired level of granularity, the EARA model uses the notion of *aggregation*. Aggregation is a special association between an entity and an aggregate. This association is also called *entity explosion* because the entity *explodes* into its components. The participating aggregate is called *child aggregate* and the entity is called *parent entity*. Basically, the aggregation means that the given entity represents the aggregate, or that the aggregate describes more details of the entity.

Entities, relationships, and aggregates may possess *attributes*. The attributes represent properties of the object and are additional means of refining software specifications.

The EARA model introduces classification of all entities, aggregates, relationships, and attributes that can exist in a software specification by defining *object types*. Each entity, relationship or aggregate is of one and only one type. The objects of the same type share a lot of properties, like aggregation capabilities for entities, roles for relationships, eligible types of components for aggregates and the names of attributes for all kinds of objects.

The object types can form *specialization hierarchies*. The hierarchy is formed when an object type is defined as a subtype. Subtypes inherit all the properties of their supertypes.

The next subsection provides a formal description of the EARA/GE model. GE stands for *Graphical Extension* and represents the graphical model portion of the EARA. It should be noted that ADL does NOT use the GE for its icon definitions due to the difficulty of associating the GE definitions with ICOLA-defined objects.

2.2.1 Notation

The EARA/GE model is based on set theory. Let us first introduce the notation that will be used in the formal definition of the model.

A set is an unordered collection of elements denoted by brace brackets, as in $\{a, b\}$ and a list is an ordered collection of elements denoted by round brackets, as in (a, b).

If A and B are sets then the following notation applies:

x ϵ A	x is an element of A
A $\cup$ B	the union of sets A and B
A $\times$ B	the Cartesian product of A and B
f:A $\rightarrow$ B	the function f from A to B

If $A = \{a, b\}$ then *powerset*(A) is the set of all subsets of A, including the empty set:

$$powerset(A) = \{ \{\}, \{a\}, \{b\}, \{a, b\}\}$$

The *powersetN*(A) is the set of all subsets of A which contain at least N elements from A. For example:

$$powerset1(A) = \{ \{a\}, \{b\}, \{a, b\}\}$$

$$powerset2(A) = \{ \{a, b\} \}$$

$$powerset3(A) = \{\}$$

Powerlist(A) is defined as the set of lists formed from all subsets of A:

$$powerlist(A) = \{(),\ (a),\ (b),\ (a,\ b),\ (b,\ a)\}$$

Powerlistn(A) is defined as the set of all lists formed from subsets of A with at least n elements. For example:

$$powerlist2(A) = \{\ (a,\ b),\ (b,\ a)\}$$

2.2.2 The EARA Model Description

A formal EARA environment definition consists of the information specified by the sets and functions listed in Table 2.1. The sets define names of various elements of the EARA model: *aggregates*, *entities*, *relationships*, *attributes*, *roles*, and *value types*. The functions specify how these elements are connected to each other in order to define the EARA model.

The *component type* function (ct) specifies the types of entities and relationships that can be components of each type of aggregate. Each aggregate type must have at least one component type.

The *aggregate type identification* function (ai) maps an entity type name to an aggregate type name. This aggregate type is also known as the *child aggregate* of a given entity. The child aggregate contains a detailed view of its parent entity. The function (ai) provides means for abstracting a set of entities and relationships into a single, higher level object called *aggregate*. The function (ai) together with functions ei and ri are used for modeling a variety of decomposition techniques offered by various software engineering methodologies.

The *entity type identification* function (ei) maps an entity type name to another entity type name in its child aggregate.

The *relationship type identification* function (ri) maps an entity type name to a set of relationships that are carried over to its child aggregate.

The *attribute association* function (aa) defines the set of attributes associated with each type of aggregate, entity, or relationship objects. A special set of attributes, called *intrinsic attributes*, are associated with every object.

The *role mapping* function (rm) specifies a list of one or more role names for each

Table 2.1: EARA Environment Definition

AgT:	A finite set of aggregate type names.
ET:	A finite set of entity type names.
RT:	A finite set of relationship type names.
AN:	A finite set of attribute names.
RN:	A finite set of role names.
VT:	A finite set of value types
ct:	$AgT \rightarrow powerset1(ET \cup RT)$ The *component* type function
ai:	$ET \rightarrow AgT$ The *aggregate type identification* function
ei:	$ET \rightarrow ET$ The *entity type identification* function
ri:	$ET \rightarrow powerlist\ (RT)$ The *relationship types identification* function
aa:	$(AgT \cup ET \cup RT) \rightarrow powerset(AN \text{ x } VT)$ The *attribute association* function
rm:	$RT \rightarrow powerlist1(RN)$ The *role mapping* function
rqd:	$(RT \text{ x } RN) \rightarrow \{\text{single, list}\}$ The *role quantitative description* function
rpt:	$RT \rightarrow powerset1(powerlist2\ (AgT \cup ET))$ The *relationship participant types* function
ag:	$AgT \rightarrow AgT$ The *aggregate generalization* function
eg:	$ET \rightarrow ET$ The *entity generalization* function
rg:	$RT \rightarrow RT$ The *relationship generalization* function

relationship type.

The *role quantitative description* function (*rqd*) specifies if a given role may be occupied by a single participant or by a list of participants.

The *relationship participant types* function (*rpt*) specifies the combinations of entity types that may participate in a given type of relationship. Each combination is a list of entity types.

The *generalization* functions (*ag*, *eg* and *rg*) map object subtypes to their appropriate supertypes for aggregates, entities, and relationships. These functions are partial functions since not all types are subtypes of other types.

2.2.3 The Graphical Extension (GE)

This subsection gives a description of the graphical interface extension to the EARA model. The extension supports the definition of a graphical representation for software objects definable in the basic EARA model. The representations are intended for the development of interactive tools that allow analysts/designers to manipulate descriptions in a specification database using a graphical workstation.

A formal definition of the GE extension is given in Table 2.2. The sets define names of various elements of the EARA/GE model: *diagrams*, *icons*, *edges*, *labels*, *handles*, *pictures*, *branches*, *nodes*, *primitives*, *properties*, *parameters*, and *value types*. The functions specify how these elements are connected to each other in order to define the EARA/GE model.

The *label association* function (lA) specifies the labels associated with an icon. The *primitive association* function (piA) specifies the graphical primitives that are components of a picture type. The *picture association* function (pcA) specifies the pictures associated with an icon. The *property association* function (poA) specifies the property-value pair associated to diagrams, icons, edges, pictures, labels, and primitives. The *parameter association* function (prA) specifies the parameters associated with pictures, labels, primitives, nodes and branches. The *handle association* function (hA) specifies the handles associated with an icon. A *handle* is defined as a part of an icon in which edges are permitted to make a connection. The *node association* function (NA) and *branch association* function (BA) specifies the nodes and branches associated, respectively, with a relationship.

The *derivation* functions (dD, iD, PcD, HD and ED) map object subtypes to their appropriate supertypes for diagrams, icons, pictures, handles and relationships. These functions are partial functions since not all types are subtypes of other types.

The correspondence between the conceptual objects and the graphical objects is shown in Figure 2.2.

Table 2.2: GE Definition

DT:	A finite set of diagram type names.
IT:	A finite set of icon type names.
EgT:	A finite set of edge type names.
LT:	A finite set of label type names.
HT:	A finite set of handle type names.
PcT:	A finite set of picture type names.
NN:	A finite set of node names.
BN:	A finite set of branch names.
PoN:	A finite set of property names.
PiN:	A finite set of primitive names.
PrN:	A finite set of parameter names.
VT:	A finite set of value types
lA:	$IT \rightarrow powerset(LT)$
	The *label association* function
piA:	$PcT \rightarrow powerset(PiN)$
	The *primitive association* function
pcA:	$(IT \cup BN) \rightarrow powerset(LT)$
	The *picture association* function
poA:	$(DT \cup IT \cup ET \cup PcT \cup LT \cup PiT) \rightarrow powerset(PoN \text{ x } VT)$
	The *property association* function
prA:	$(PcT \cup LT \cup PiT \cup NT \cup BT) \rightarrow powerset(PrN \text{ x } VT)$
	The *parameter association* function
hA:	$IT \rightarrow powerset(HT)$
	The *handle association* function
NA:	$EgT \rightarrow powerlist(NN)$
	The *node association* function
BA:	$EgT \rightarrow powerlist(BN)$
	The *branch association* function
dD:	$DT \rightarrow DT$
	The *diagram derivation* function
iD:	$iT \rightarrow iT$
	The *icon derivation* function
eD:	$EgT \rightarrow EgT$
	The *edge derivation* function:
pD:	$PcT \rightarrow PcT$
	The *picture derivation* function
hD:	$HT \rightarrow HT$
	The *handle derivation* function

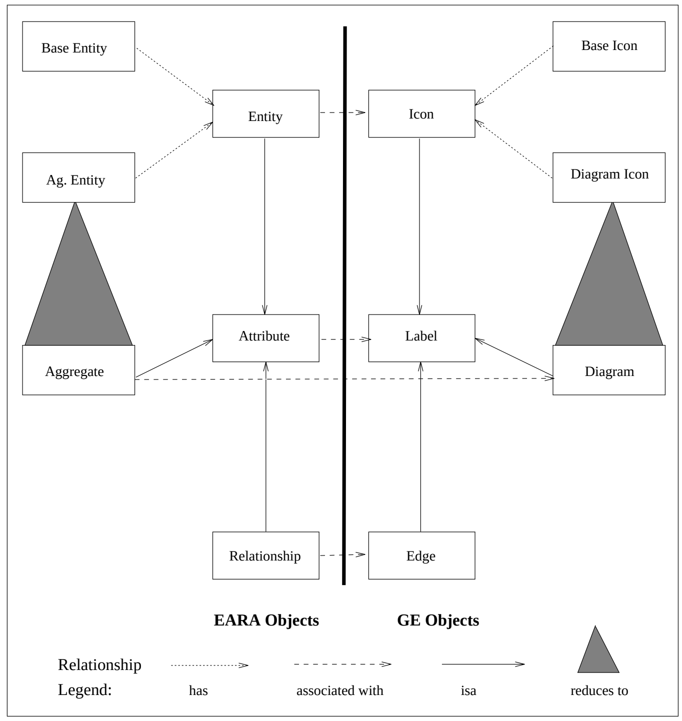


Figure 2.2: Mapping between the EARA and GE Objects

2.3 Specification Database

The EARA data types defined in Section 2.2 can be instantiated with the entities, relationships, and aggregates of a software system. A specification database is an instantiated method (environment).

In Metaview, a graphical editor is used to create the specification database. The specification database forms the input for Metaview tools. The database query support, the *Project Daemon*, provided by Metaview allows Metaview tools to gather specification information from the database. The Automatic Diagram Layout (ADL) Tool is an example of a Metaview tool.

CHAPTER 3

Related Work

While much work has been done to develop graphical editors, there has been some interesting work done in the area of automatic software diagram layout. Diagram layout, manual or otherwise, essentially reduces to the two problems of placement and routing. The placement problem refers to choosing the locations of diagram icons. The routing problem involves determining paths for the edges which connect the diagram's icons.

In this chapter, we briefly examine some of the methods which have already been developed. A few approaches implementing a combined placement and routing strategy are reviewed. The combined placement and routing strategy essentially merges the two problems. For example, one would place and then route, and iterate on this procedure, potentially placing and replacing the same icon several times. To this end, a brief treatment of MONDRIAN, a system which is dedicated to the production of data flow diagrams, is presented. This treatment is followed by an examination of the graph-theoretic approach, with a brief examination of GIOTTO, a system dedicated to the generation of Entity Relationship and Data Flow diagrams, and DAG, a program to draw directed graphs. The graph-theoretic approach views the desired diagram as a graph whose nodes represent entities and whose edges represent relationships [6] [24].

The chapter then concludes with a brief explanation of Lee's Algorithm which is not really an automatic diagram layout approach but is an automatic edge routing approach. Lee's algorithm is the basis for many of the approaches to automatic layout in the current literature.

3.1 MONDRIAN — A Diagram-Specific Layout Algorithm

MONDRIAN [20] is a system that automatically generates Data Flow diagrams. This is an approach that is very specific to Data Flow diagrams and is not a general layout tool. The algorithms are derived from those used for automatic circuit layout design, but with some important modifications to ensure that there is sufficient space between icons to allow room for labels and produce a readable diagram. MONDRIAN takes as input a specification database describing, textually, the data flows, data stores, processes, and interfaces. This information is then represented by a directed graph, which is used to build the layout. Placement of the icons depicting data stores, processes, and interfaces and routing of the edges depicting data flows are handled in two separate steps. The resulting diagram features fully-labeled icons and edges, as well as arrowheads on edges to indicate direction of flow. All edges follow a grid orientation, and so only 90° bends are allowed. Replication of interface and data store icons is allowed, in accordance with the Gane and Sarson [10] Data Flow diagram convention. This replication helps to minimize edge crossings, arc lengths, and bends. Diagrams generated by MONDRIAN can be modified via a graphical editor which the system provides.

3.1.1 General Architecture of MONDRIAN

This section describes the major components of the MONDRIAN system. MONDRIAN bases its design on the adaptation of a circuit design automation pair-linking placement algorithm and a predetermined endpoint routing algorithm.

Figure 3.1[20] shows the high level structure of MONDRIAN. MONDRIAN gets its information from an SPSA (Simple Problem Statement Analysis) database and

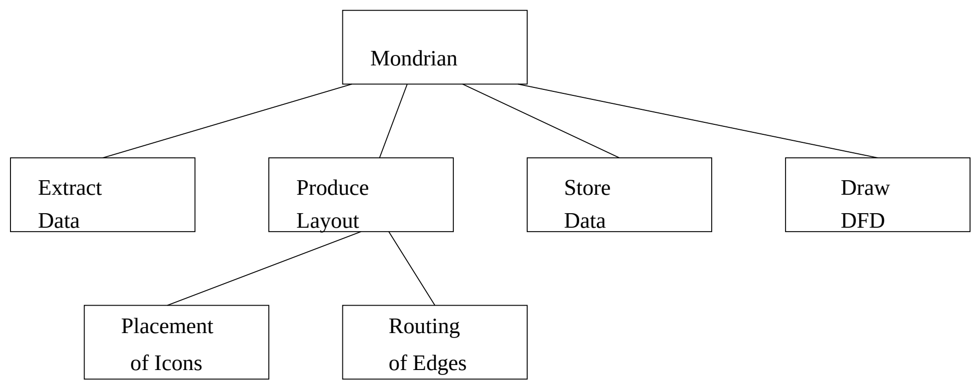


Figure 3.1: High-level Structure of MONDRIAN

stores it in its internal data structures for use by the placement and routing algorithms. The data structures include an adjacency list that records information about objects, edges, and relationships and three grid structures that are used to assist in and record the layout process. The adjacency list is used to represent the system flow information contained in the requirements specification SPSA database. The adjacency list also records the layout as it is determined. Each flow is considered to be a ternary relationship of the form "x flows from y to z" where x is a data flow and y and z are objects (processes, data stores, or interfaces). In this form of ternary relation, y is the originating object, and z is the terminating object.

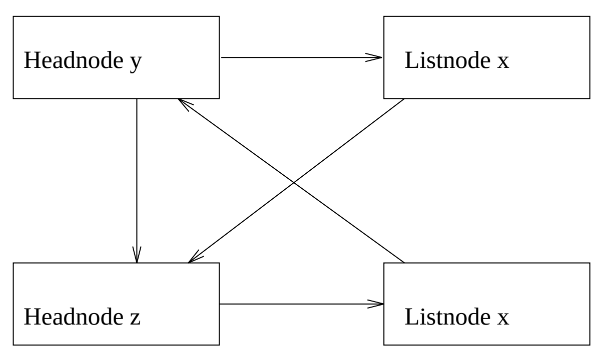


Figure 3.2: MONDRIAN's Adjacency List Structure

Figure 3.2[20] shows the adjacency list structure for the generalized ternary relationship. In the adjacency list, a headnode represents an object and a listnode represents a data flow. Each headnode has a linked list of listnodes that represent all data flows incident to it. Each listnode has pointers to the other headnode involved in the relationship and to the listnode in the second headnode's list that represents the same data flow. This allows the placement of objects to be made in any order and connections to be made as each object is placed.

Each headnode contains information about object identity (e.g., name and type), object position (e.g., number of replications and location of each icon), and object connectivity (e.g., indegree and outdegree). The object identity is used both in producing the layout and for accessing the SPSA database. The object position(s) is used solely in producing the layout. Connectivity is used by the placement algorithm. A listnode contains information about the identity of the data flow, the direction of flow, and to which instance of a replicated icon the flow is connected. Each listnode also contains pointers to a linked list of pathnodes and the pathnode selected for the data flow to follow. The pathnode contains specific details about the path, such as start and end points, position and type of corners, position of a label, which axis it starts on, and what direction it travels on the y axis.

The adjacency list is constructed by first creating and initializing a headnode for each process, data store, and interface object in the SPSA database. Then for each headnode, each system flow relationship is extracted from the database. For each of these relationships, a listnode is allocated, initialized and linked into the list for the particular headnode. A link is established from the listnode to the terminating headnode (that is, if the listnode is in the set belonging to the originating headnode, a link to the terminating headnode is made).

To record the placement and connection of objects, three grid systems are used. The first, the main grid, is used to record the placement of objects. The information contained in this grid is the object's name, type, identification number, and the number of times the object is replicated. The second grid, the subgrid, represents a 5 x 5 subdivision of each square in the main grid. The subgrid is used to record the details of the connections (data flows) between objects. A template is superimposed, starting from the top left corner of the main grid, which has certain positions which can and cannot have icons placed in them. The template has a repeating 4 x 4 pattern in which there are four possible positions to have icons placed. This template was determined empirically by the designer of MONDRIAN. The template allows a maximum of 16 connecting positions for each placed icon. This 4 x 4 template was compressed to create the third grid structure, named the workspace, by dividing the main grid into segments of two squares by two squares such that each segment contains one allowable position for an icon according to the template. This third grid, the workspace, is one-quarter the size of the main grid and is used to keep track of the possible connecting nodes. Four distinct segment types are thus formed by this division. An array that contains the relative x and y coordinates required to reach each of the 16 possible positions in the workspace grid is created for each of the segment types. The workspace square contains an indicator of its type and a value used as an index into the corresponding alternatives array.

These preceding data structures are used directly by MONDRIAN's placement and routing algorithms given below.

3.1.2 MONDRIAN's Placement Algorithm

The proper placement of icons is critical in the production of easy-to-read data flow diagrams. This section briefly describes the placement algorithm used by MONDRIAN. The placement algorithm is an adaptation of the design automation pair linking algorithm [13]. It begins by choosing, as a nucleus, the pair of nodes that has the largest sum of common weighted signal sets, where a signal set is a set of points (icons) that is to be connected by one wire (arc). The next node and all remaining nodes to be placed are selected on the basis of pair connectivity (i.e., the number of arcs that connect two nodes). The unplaced node that has the highest pair connectivity with a placed node is selected and positioned as close as possible to the placed node. These preceding steps are outlined in the following algorithm:

1. Determine connectivity of each pair of nodes.
2. Select initial pair of nodes for placement based on weighted signal sets.
3. Position initial pair of nodes.
4. Repeat steps 5 and 6 until all nodes are placed.

5. Select next node for placement based on highest pair connectivity
6. Place next node.

3.1.3 MONDRIAN's Routing Algorithm

This section briefly describes the approach to DFD routing. MONDRIAN's predetermined endpoint algorithm routes each arc in the order determined by an ordering routine and considers only how the path affects those arcs that have already been placed. The steps of this algorithm are outlined as followed:

1. Order arcs on basis of number of intersections.
2. Repeat steps 3 through 5 for each arc in the order determined in step 1.
3. Find all possible paths for arc.
4. Rank paths.
5. Assign highest ranked path to subgrid (grid number two).

3.2 GIOTTO

GIOTTO [1] is a diagram drawing tool which currently generates Entity Relationship and Data Flow diagrams based on the graph-theoretic approach but its developers hope to generalize it to produce a wider variety of diagrams. GIOTTO allows the user to set up certain aesthetic criteria and semantic constraints on any of its diagrams to ensure readability. The aesthetic criteria are:

- CROSS: minimization of edge crossings
- BENDS: minimization of edge bends; this assumes a grid standard as opposed to a straight-line standard
- LENGTH: minimization of edge length

- AREA: minimization of area of diagram
- VERT: nodes at the same level, in a hierarchical structure, are placed on a horizontal line
- SYMM: parents are centered above children in a hierarchy

These aesthetic criteria define a diagram's layout regardless of the information the diagram conveys. Yet a diagram's appearance also should portray the nature of relationships between entities in the specification database that the diagram is meant to represent. That is, entities that are grouped together in the database should be drawn together in the diagram in a way that clearly shows the nature of their relationships. To accomplish this, semantic constraints are placed on a diagram. These semantic constraints are:

- CENTER: placement of certain vertex/vertices at the center of a diagram
- EXTERN: placement of certain vertex/vertices on the external face of a diagram
- NEIGH: grouping of certain vertices together
- STREAM: placement of a certain sequence of vertices in a straight line
- SHAPE: drawing a subgraph with a specified shape
- CMAX, BMAX, LMAX: setting a maximum number of edge crossings, bends or a maximum length for a specified edge

In GIOTTO the user may apply the aesthetic criteria and semantic constraints locally or globally. That is, they may refer only to a part of a diagram or to the whole, respectively. The concepts of aesthetic criteria, operating independently of the underlying specification database, and semantic constraints, which help to express visually the information contained in the database, are used explicitly in the algorithm to generate the diagrams. The algorithm which GIOTTO employs is as follows:

1. Start with a conceptual schema (a list of what relates to what); effectively a specification database.

2. Model this into a conceptual graph that connects entities appropriately.

3. Produce a planar representation of the conceptual graph that describes the topology of the embedding. If the graph is not planar, crossings are replaced by dummy vertices to fake planarity. This step uses the aesthetic criterion CROSS and semantic constraints CENTER, EXTERN, NEIGH, SHAPE, and CMAX.

4. Produce an orthogonal representation in which the only angles allowed in the edges are right angles, which is the grid standard. This gives the shape of the eventual diagram without regard to dimensions. This step uses the aesthetic criterion BENDS. If hierarchical subgraphs are present then the aesthetic criteria VERT and SYMM are also used. The semantic constraints SHAPE, STREAM, and BMAX are used as well.

5. Compact to get the grid representation, which is the skeleton of the final diagram, properly sized. This step uses the aesthetic criteria AREA and LENGTH, as well as the semantic constraint LMAX.

6. Flesh out the skeleton diagram with appropriate icons to produce the final diagram.

GIOTTO can stand alone or provide support to a graphical editor. GIOTTO is impressive for its speed and its capability to draw both Data Flow and Entity-Relationship diagrams. It does have drawbacks, however, in that it does not replicate icons to reduce edge crossings in Data Flow diagrams, and has no labels on edges.

3.3 DAG — A Program that Draws Directed Graphs

Gansner et al. [11] developed a prototype tool for producing layouts of directed acyclic graphs (DAGs for short).

Directed graphs have many applications in computing, such as describing data structures, finite automata, data flow, procedure calls, and software configuration dependencies. A picture is a good way to represent a directed graph. It is seldom easy to understand much about a graph from a list of edges, but with a picture one can quickly find individual nodes, groups of related nodes, and trace paths in the graph.

DAG is a program that draws directed graphs from edge lists. DAG works well on acyclic graphs and other graphs that can be drawn as hierarchies. It reads descriptions in a concise language and makes POSTSCRIPT documents of the picture. The concise language is simple to use. For example, Figure 3.3 illustrates a simple directed graph specification and its corresponding diagram. The language syntax and semantics are laid out in much more detail in Gansner [11].

```
.GS
edge from a to b;
edge from b to c;
edge from c to a;
.GE
```

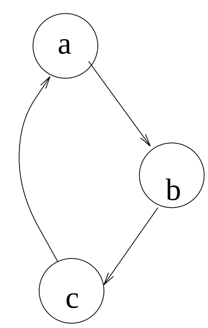


Figure 3.3: DAG Example Specification and Corresponding Diagram

3.3.1 DAG's Drawing Algorithm

To design algorithms for drawing graphs, we need to define what makes a good drawing. Three desirable properties for drawings of directed graphs follows:

P1. The drawing should reveal the partial order implied by edges of the graph. Nodes are placed in ranks so that edges point downward.

P2. The drawing should not be cluttered. Edge crossings are undesirable because they suggest a connection that is not really present in the underlying abstract graph specification. Edges should also not intersect nodes that are not the edge's endpoints.

P3. The drawing should reveal the relationships implied by edges by placing adjacent nodes in the graph close together.

Unfortunately, these goals conflict. For instance, the placement of nodes in ranks or tiers may make it impossible to avoid edge crossings. Also, the minimization of edge-crossings in a layout is, according to Gansner[11], computationally intractable. Thus, DAG implements heuristics which run quickly and make good layouts in common cases.

DAG has four passes. The first pass attempts to satisfy P1 and P3 by finding an optimal rank assignment. The optimal rank assignment problem is to assign integer ranks to nodes such that the sum of the weighted edge lengths is minimized. The length of an edge here is taken as the difference in the ranks of its head and tail nodes. Following rank assignment, DAG creates dummy nodes where long edges cross ranks. The placement of dummy nodes determines edge crossings and is also an aid to drawing splines for long edges.

The second pass orders nodes from left to right within ranks. This pass tries to reduce edge crossings (P2) subject to the constraints on node order implied by flat edges (P1). It produces less crossings by reducing the effects of widely spread nodes. It accomplishes this by using a heuristic called the generalized median. Informally, the generalized median of a node with respect to a generalized rank is defined as follows. When the number of neighbors of a node in the concerned adjacent rank is odd, the generalized median is simply the median position of its neighbors. When the number of neighbors is even, the generalized median is in between the left and the right median positions. The exact value is tilted toward the side where neighbor nodes are more tightly packed. The generalized median heuristic tends to work better if the initial order of nodes is good.

The third pass assigns absolute coordinates to nodes, respecting the ordering determined by the previous pass, thus adhering to P3. The optimal assignment can be formulated as a set of linear constraints and an objective function involving absolute values. This can be transformed into a linear program by a standard technique that involves introducing extra variables.

The fourth, and final, pass finds the spline control points for edges. There are several cases to consider, including self-edges, edges between nodes on the same rank, adjacent ranks, or non-adjacent ranks, possibly in the presence of multi-edges. Short edges are easy to draw. Long edges, or edges that have endpoints on non-adjacent ranks, are more difficult since they may pass near other nodes and change direction.

The dummy nodes are created with a height a little larger than the highest non-dummy node in the same rank, and width usually twice the default node separation. This is big enough to make a spline inside the boundaries of the dummy node, if necessary. On the other hand, if there is space next to the dummy node, it is often preferable to encroach on it to draw a smoother spline. Dummy nodes are deleted if it is determined that routing an edge through them is not required for whatever reason.

3.4 Lee's Algorithm – The Basis for Routing

A version of breadth-first search known as Lee's algorithm [16] was applied to the printed circuit board autorouting problem in the early 1960's, and is still the basis of some autorouters. Unfortunately, Lee's algorithm suffers from a behavior inherent in the breadth-first search technique which limits its application to problems of relatively small size. As the distance between the source and target cells increases by a factor of n, the number of cells processed by Lee's algorithm (and therefore the processing time) increases by the square of n.

When search algorithms are used to solve the circuit routing problem, they typically operate in two phases [2], and treat the printed circuit board's two dimensional surface as a matrix of cells. Both Lee's algorithm and the A* algorithm, of section 4.8, use two phases to route a path. The first phase starts at the source cell and searches for the target cell, usually by going in several directions at the same time. While searching, an auxiliary data structure will be built which keeps track of how each cell was reached. Once the target cell has been found, the first phase is finished, and the second phase is started. However, if the first phase exhausts all possibilities without reaching the target cell, then no route exists between them, and there is no reason to do the second phase.

In the second phase, the auxiliary data structure is used to trace the route from the target cell back to the source cell. The second phase is identical for both Lee's and the A* search algorithms. But the first phase is different, and it is this difference that gives these algorithms different behaviors.

The main data structures used in the first phase are the *OpenQueue* and the *ClosedSet*. These structures are used to hold cell coordinates. Since a cell's coordinates uniquely identify that cell, we'll say that the *OpenQueue* and *ClosedSet* contain cells. Cell coordinates will be represented as, for example, *r2c3* for the cell at row 2, column 3. Initially, the *OpenQueue* contains the source cell and the *ClosedSet* is empty. The first phase is a loop which removes an element from the head of the *OpenQueue*, puts it in the *ClosedSet* (which indicates that it has been searched), and checks to see if it is the target cell. If it is, the first phase is done. Otherwise, the neighbors of the cell (the cells that are adjacent to it) are placed on the *OpenQueue*, and the loop continues. As we will see in section 4.8.1, the essential difference in Lee's and the A* search algorithms is the order in which the neighbors are placed on the *OpenQueue*.

Lee's algorithm works by processing a first-in-first-out (FIFO) queue of open cells (cells that have been reached, but not yet searched). Initially, the *OpenQueue* contains only the source cell. A cell is removed from the head of the *OpenQueue*, placed in the set of closed cells (cells that have been searched), checked to see if it is the target cell, and if it is not, its neighboring cells are placed at the tail of the *OpenQueue*. Note that if a neighboring cell is classified as blocked, then that neighboring is not placed on the *OpenQueue*. Blocked means that there is something in the neighboring cell that a path can't be routed through. Neighboring cells which have already been reached are ignored. If a cell's coordinates are on the *OpenQueue* or in the *ClosedSet*, then it has been reached. Otherwise, it has not been reached. This continues until the target cell has been found, or the *OpenQueue* is empty, in which case the target cell cannot be reached from the source cell. Lee's algorithm follows.

The following data structures are also used in Lee's algorithm. *StartCell* is the start node, *TargetCell* is the goal node. *OpenQueue* is an ordered list of nodes (ordered by arrival time; nodes enter at the tail and leave at the head), also known as an abstract data type (ADT) called a queue. The operations allowed on the OpenQueue ADT are:

- QInsert(Q,element) — insert *element* into Q;
- QDelete(Q) — delete the first element from Q and return with its value;
- QEmpty(Q) — return true if Q has no elements, and return false if there is at least one element in Q.

ClosedSet is a set of nodes (order doesn't matter). In general, nodes that need to be searched are put on *OpenQueue*. As they are searched, they are removed from *OpenQueue* and put in *ClosedSet*. *PredCell* ("Predecessor Cell") is defined for each node, and is a list of "came from" indications, so when we finally reach *Targetcell*, we traverse *PredCell* to construct a path to *StartCell*. The *PredCell* data structure can be implemented as a simple two-dimensional vector in which each element of the vector contains an ordered pair of numbers representing coordinates to another element of the *PredCell* vector. The algorithm starts with *PredCell[TargetCell]*. *PredCell[TargetCell]* are the coordinates of the cell adjacent to the *TargetCell*. The *PredCell* value for *PredCell[Targetcell]* would then be used to to find the next path cell until *StartCell*'s coordinates are eventually reached. Therefore, traversing from *TargetCell* back to *StartCell* via the *PredCell* data structure is basically a singly-linked list traversal. Note that in Figure 3.4 the arrows on the diagram represent the *PredCell* data structure.

Lee's Algorithm: (* Breadth-First Search *)

Declarations:

StartCell, TargetCell - The source and destination cells for the search

OpenQueue - The main priority queue for the search

PredCell - The predecessor cell array which is used to get the coordinates
  of the bends in a successfully routed edge

NeighborSet - This is the set of the four immediate neighbors
  of the current node being searched

neighbor - One of the neighbor nodes in NeighborSet

current - This is the current node being searched

CurrentPredCell - This variable is used to assist in the traversal of PredCell

Step 1. (*Initialize the data structures*)

```
    QInsert(OpenQueue,StartCell)
    ClosedSet ← {}
    PredCell[StartCell] ← NULL, found ← FALSE
Step 2. (*perform phase 1 — breadth first search*)
    WHILE not QEmpty(OpenQueue) and not found DO
        current ← QDelete(OpenQueue)
        Closedset ← ClosedSet ∪ {current}
        IF current = TargetCell
        THEN found ← TRUE (* we're done *)
        ELSE (* continue search through node current *)
            NeighborSet ← {set of neighboring nodes of current}
            FOR each neighbor in NeighborSet DO
                IF neighbor is not on OpenQueue or in ClosedSet or
                    neighbor is not blocked
                THEN
                    PredCell[neighbor] ← current (* remember where we came from *)
                        QInsert(OpenQueue,neighbor)
Step 3. (*perform phase 2 — retrace path from Targetcell back to StartCell*)
    IF found THEN
    (* traverse the PredCell data structure. *)
        CurrentPredCell ← PredCell[Targetcell]
        WHILE CurrentPredCell ≠ StartCell DO
            CurrentPredCell ← PredCell[CurrentPredCell]
        ELSE
            Inform user that Targetcell cannot be reached from StartCell
```

The behavior of Lee's algorithm while searching for a path between a source cell S (*StartCell*) (r5c5) and a target cell T (*Targetcell*) (r7c7) is shown in Figure 3.4. Lee's algorithm does not specify the order in which neighboring cells are placed on the open queue, but the compass directions north, east, south, and west will be used. This order tends to produce traces with a minimal number of turns.

In Figure 3.4(a), the source cell (r5c5) has been searched, and its four neighbors have been placed on *OpenQueue*. The arrows indicate the direction from which each cell was reached, and correspond to the *PredCell* data structure. After the first four cells on the *OpenQueue* have been reached and moved to the *ClosedSet*, the configuration in Figure 3.4(b) is reached, where there are eight cells on *OpenQueue*. Once these eight cells have been searched, the configuration in Figure 3.4(c) is reached. Now *TargetCell* (r7c7) is very close to being put on the *OpenQueue*, thus, a solution is imminent. When r7c7 is searched, it will be recognized as *TargetCell*, and the *PredCell* data structure will be used to construct a trace back to the source cell; refer to the bold arrows of Figure 3.4(c).

The search progresses outward from the source cell in all directions, like the ripples in a pond when you throw a pebble into the water. If we double the size of the problem, the number of cells searched (and therefore the processing time) will be roughly four times as large, and if the size of the problem is tripled, the number of cells searched will be roughly nine times as large. Thus, the behavior of Lee's algorithm is quadratic in the size of the problem, which makes it infeasible for very large routing problems.

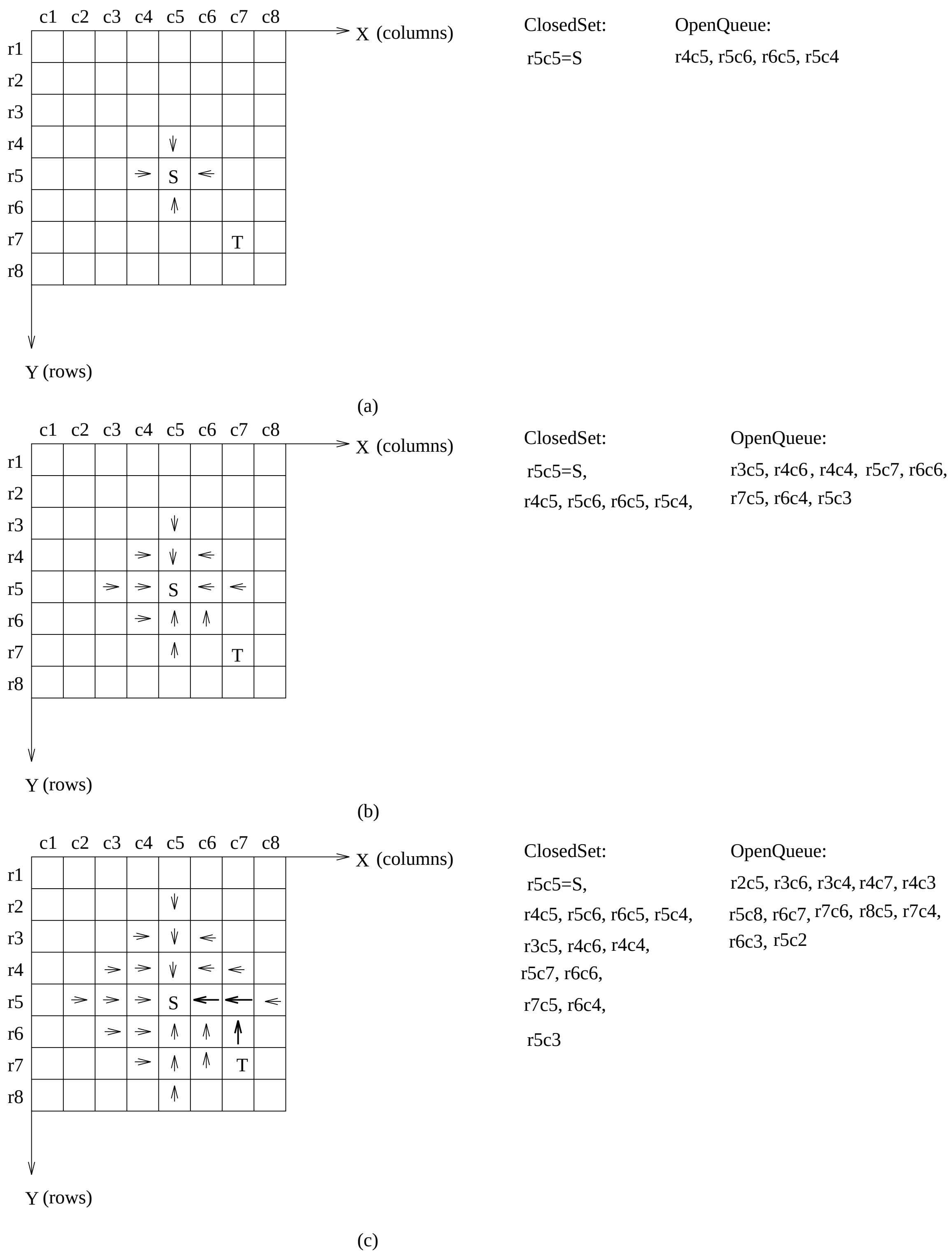


Figure 3.4: A Sample Run of Lee's Algorithm

CHAPTER 4

ADL System Design

4.1 Introduction

In this chapter, a novel approach to automatic software diagram layout will be discussed. The approach is incorporated within a software tool called the Automatic Diagram Layout(ADL) System. The ADL system is a Metaview-based software development system. The main goals of the ADL system prototype development are:

- First, the ADL system is to provide a methods engineer with a diagram layout tool. A methods engineer specifies, or models, most, if not all, of a diagram type's characteristics. The ADL system's basic premise is to give the methods engineer as much control over the layout as possible. ADL instantiates the graphical model with Metaview specification database information. The modeling of the diagram occurs at the Method/Environment level in Metaview. The layout of the diagram occurs at the user level of Metaview. The diagram is a visualization of a specification database. A specification database possesses the actual entity, relationship, aggregate, and attribute instances which are present on the diagram.

- ADL is a research prototype. The second and equally important goal of the ADL system is to provide a basis for evaluating the isolated placement and routing strategy and the viability of the placement algebraic approach to diagram layout.

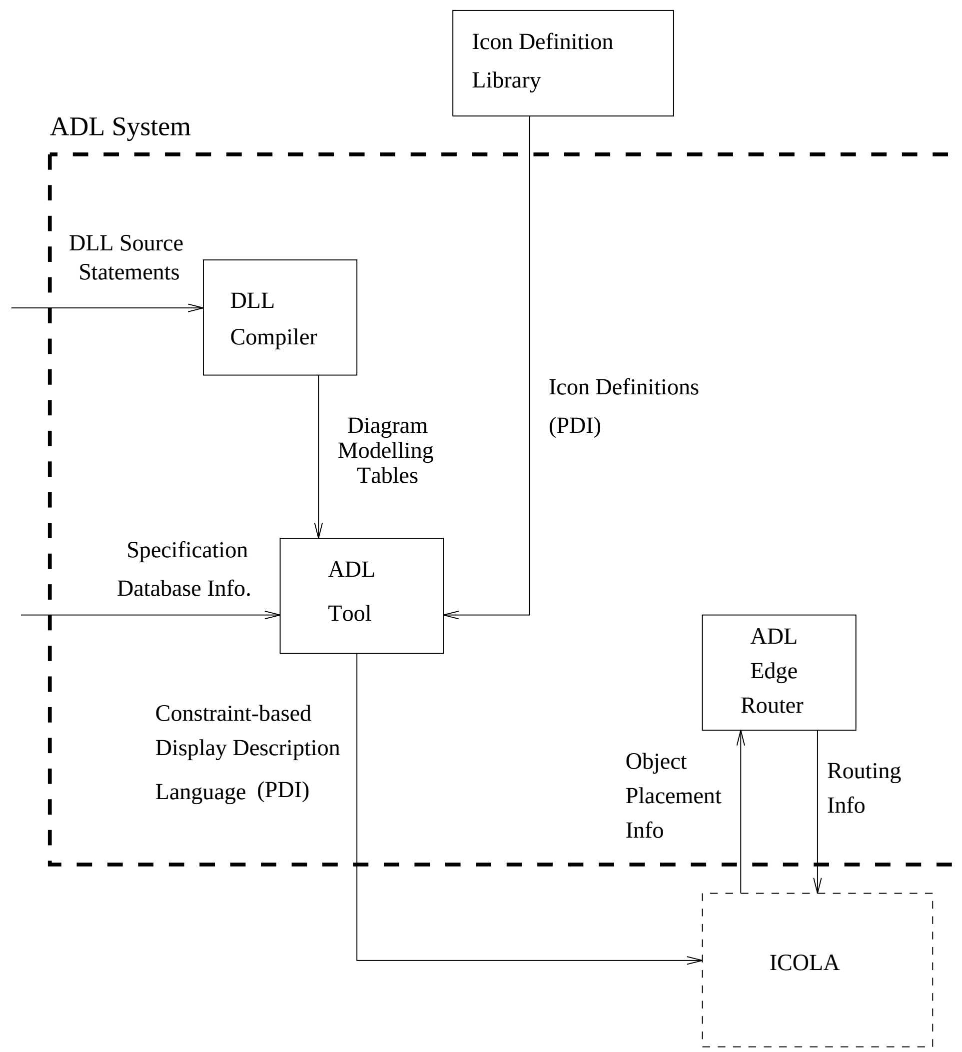


Figure 4.1: Automatic Diagram Layout System

The ADL system is composed of three components as illustrated in Figure 4.1. This figure was reproduced from Chapter 1. These components are:

- Diagram Layout Language (DLL) compiler — This compiler produces a diagram type's modeling tables.

- ADL Tool — This tool uses a diagram type's modelling tables to generate code for the object layout platform, ICOLA.

- ADL Edge Router — This router determines the paths for edges on a diagram.

A methods engineer encodes the the layout rules of a diagram type in a DLL program. A DLL program is translated from its high-level form by the DLL compiler into diagram modeling tables which form one of the inputs to the ADL Tool of Figure 4.1. The modeling tables are produced by the DLL compiler for a given diagram type once. The ADL Tool is responsible for generating diagram layouts for each specification database chosen by a system developer. The ADL Tool processes its inputs (diagram tables, icon definitions, specification database), and produces, as its output, an object language instantiation of the DLL-modeled diagram type. The object language forms the main input to a graphical object layout platform, ICOLA [15]. This platform accepts this object language and generates the actual diagram. The routing of the edges by the ADL Tool is merely the generation of the appropriate input commands to ICOLA stating what the participant icons are for routed edges. The routing of edges on a given diagram is performed within the ICOLA platform via the A* routing strategy (Tanimoto [26]). This strategy is explained in Section 4.8.

The basic premise for the design of the ADL system was to decouple the placement problem from the routing problem. This thesis work is an example of an isolated placement and routing approach. Isolated placement and routing, simply stated, is placing the icons and then, after all placements are done, routing the edges between the related icons without any attempt to reiterate on either placement or routing after both tasks were accomplished. Why would we want to isolate the placement problem from the routing problem? The only reason for decoupling placement and routing is because the placement platform chosen, ICOLA, excluded the possibility of being able to route concurrently with placement. The edge routing isn't invoked until all icons have been completely placed. Thus, basing the design and implementation of the ADL system on the ICOLA platform prevents the tight coupling of placement and routing. Using ICOLA as the platform was an advantage at the design stage because ICOLA performs all of the graphic work and much of the placement constraint solving. This left the problem of generating the relative location constraint code, the placement algebraic statements, which ICOLA accepts in order to layout the diagram. The ADL Tool is responsible for generating the input to ICOLA.

The edge routing was also a problem that needed a solution. The detailed edge routing uses the A* search algorithm. Once ICOLA places all of the icons it will then employ the A* routing strategy of Section 4.8 to find paths for all of the diagram's edges.

The rest of this chapter expands on the introduction. Section 4.2 contains a sample formal Metaview method definition, with a corresponding specification database, and a possible diagrammatic layout of that specification. Section 4.3 focuses on the placement algebraic approach to the placement of icons for automatically generating the aforementioned possible layout. Section 4.4 provides a detailed treatment of how placement algebra is incorporated into the ADL system via the Diagram Layout Language. The Diagram Layout Language is introduced in Section 4.5 and formally defined in Section 4.6. The ADL Tool is discussed at a high level in Section 4.7. Finally, the chapter concludes with a discussion of the solution to the routing problem via the A* search algorithm.

4.2 A Metaview Example

4.2.1 Sample Method Definition

Recall that Metaview has three layers; the meta level, the environment or method level and the user level. As discussed in Section 2.1, a methods engineer can model a particular method by using the Environment Definition language (EDL). An EDL program gives a conceptual model of a particular method. The code segment in Figure 4.2 represents a Metaview environment level model of a software method. Specific programs written in EDL represent software methods at the environment level. Instantiations of the aggregate, relationship, and entity types in an EDL program represent user level specification databases. They are instantiations of Metaview environment-level methods.

The EDL code[9] provided in Figure 4.2 is a portion of the Ward and Mellor Real Time Method. In this method, the system developer describes a problem domain by constructing a set of formal models. These models are:

- the *Transformation Model* to specify the processing required for transforming inputs to produce the system outputs. This model is an extension to the Data Flow Model used in the Structured Analysis and Design methodology of Gane and Sarson [10] to handle control as well a data transformations.

- the *State-Transition Model* to specify the system's response to events over a course of time.

- the *Data/Entity-Relationship Model* to define the conceptual entities (objects) of the data domain and the relationships between those entities.

The example used here focuses in on the *Data Model* portion of the Ward and Mellor methodology because of its simplicity. The *Data Model* or *Entity-Relationship Model* definition presented in Figure 4.2 is used as an example of how the ADL system uses the Diagram Layout Language and placement algebra to model a diagram layout. The EDL specification concisely shows the main entity and relationship types used in this example. It describes the classifications of stored data and the relationships between them. The Data Model is similar to the *Entity-Relationship* model proposed by Chen [4]. The Data Model contains *object types*. Object types are linked with each other by a set of *relationships*. The definition of the method begins with a definition of the *aggregate type* in which all entity and relationship types are *components*. The aggregate type is the conceptual version of a diagram type. There are two types of entities in the EDL-defined model: *object type* and *relationship*. Note in this example that one entity type is called *relationship*. This is simply because in the *entity-relationship* method, relationships are represented by icons as well as edges. The Data Model contains three types of relationships: *binds*, *association*, and *is supertype*. The *binds* relationship type shows the actual connections between the Data Model relationships and object types. The *association* relationship type combines an object type and a Data Model relationship (i.e., an EDL entity type) together as a logically single object. The *is supertype* relationship type is used to show the supertype of the object types bound by a supertype/subtype hierarchical relationship. The EDL code defining the relationship types involves clauses which define eligible participants. The *ROLES* clause defines the functionality of the participating entity types. The

PARTICIPANTS clause defines the eligible participant entity types for each specified role.

For further details on the Environment Definition Language and the Ward and Mellor Real Time Method refer to Gadwal et al. [9].

4.2.2 Sample Metaview Specification Database

Metaview specification databases are currently created and updated using a graphical editor called MGED (Metaview Graphical EDitor) and stored in Quintus Prolog form on disk. The following specification database illustrates an instantiation of the environment level method definition defined in Section 4.2.1. This specification database is at the user (or system developer) level of Metaview. The method that it instantiates is at the environment level of Metaview.

The specification database will be used to introduce the Diagram Layout Language (DLL). The Automatic Diagram Layout (ADL) system, via the ADL Tool, reads in a Metaview specification database and then proceeds to lay it out according to defined placement algebraic rules encoded in the diagram modeling tables. A possible ADL-generated layout of the given specification will be illustrated.

The following specification database is presented using a Simple Statement Language (SSL) as defined in McAllister [18].

```
BEGIN data_schema book_classification;
    object_type book;
    object_type fiction;
    object_type non-fiction;
    relationship relationship0
        (relationship_name = "subtype/supertype");
    binds (relationship0, fiction);
    binds (relationship0, non-fiction);
    is_supertype (book, relationship0);
END book_classification;
```

```
/*****************************************************************/
/*      Data Schema - Entity-Relationship                        */
/*****************************************************************/

AGGREGATE_TYPE data_schema  IS_A universal_agg
    COMPONENTS (object_type, relationship, association, is_supertype, binds);

ENTITY_TYPE object_type   IS_A universal_entity
    ATTRIBUTES (meaning: text, composition: text, objecttype: text);

ENTITY_TYPE relationship  IS_A universal_entity
    ATTRIBUTES (relationship_name, string);


RELATIONSHIP_TYPE binds  IS_A universal_rel
    ROLES (relationship, entity, dummy)
    PARTICIPANTS
      (relationship, object_type, dummy);

RELATIONSHIP_TYPE is_supertype  IS_A universal_rel
    ROLES (supertype, the_rest)
    PARTICIPANTS
      (object_type, relationship);

RELATIONSHIP_TYPE association IS_A universal_rel
    ROLES (object, relationship)
    PARTICIPANTS
      (object_type, relationship);
```

Figure 4.2: EDL Definition of the Entity-Relationship Model

4.2.3 Possible ADL-generated Layout

The specification database of Section 4.2.2 can be laid out in many ways by ADL. For example, Figure 4.3 depicts one way in which ADL could layout the specification. The entity type *object_type* is represented by a rectangular icon in which the the entity type instance's name attribute is its label. The entity type *relationship* is represented by a diamond-shaped icon in which the label is given by the value of one of its attributes. The *binds* and *is_supertype* relationship types are represented by unlabeled, undirected edges. The labeling which appears on the diagram is for the benefit of the reader to note which edges correspond to which relationships. The *book* entity icon is above the *subtype/supertype*, *fiction*, and *non-fiction* entity icons to convey the *is_supertype* relationship, whereas the *fiction* and *non-fiction* entity icons are placed in line with each other to show they are both equally important. Figure 4.3 is not necessarily exactly what ADL would generate but the figure captures the essence of what ADL is attempting to convey in a graphical format of what the specification conveys in a textual format.

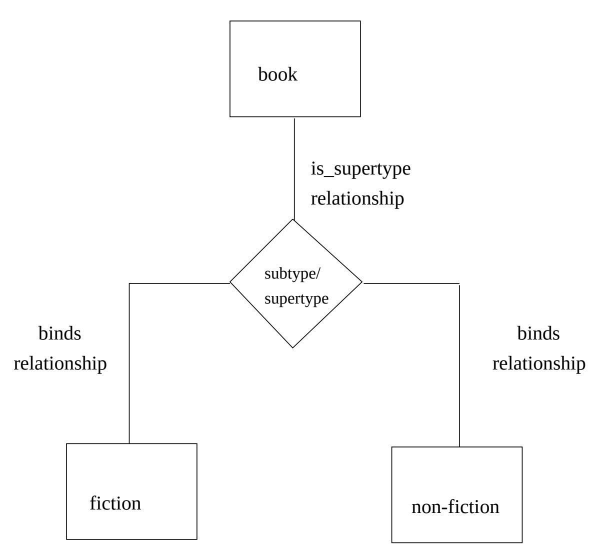


Figure 4.3: Possible ADL-generated Diagram Layout

4.3 Icon Placement and Placement Algebra

The general placement problem consists of determining the optimal positioning of interrelated objects to locations contained in a given area subject to a set of constraints. If one considers the aforementioned objects to be icons then the icon placement problem is the general placement problem. A well known mathematical problem known as the *quadratic assignment* problem, which is known to be NP-complete [12], reduces [13] to the general placement problem. Therefore, icon placement is an NP-complete problem. A method, therefore, was developed which dramatically prunes this space heuristically. This heuristic is placement algebra. The idea was first presented in Tremblay [20] as a future concept. This thesis takes that basic notion and carries it one major step forward. The notion of placement algebra was also presented in Flemming [8], Kundu [14], Mallgren [17], and Mitchell [19]. The word heuristic is used because placement algebra is a notion of relative, not absolute, positions. When relative positions are used then there is no guarantee that the relative positions are the best positions but it merely represents a very good estimate in most situations. The placement algebra solution presented in this thesis is a subset of placement algebra. However, even if the subset presented here was expanded to include every conceivable placement algebraic operation, the solution would still be heuristic because it will always lack the ability to give each developer exactly what he/she views as perfectly readable.

The placement algebra approach is a heuristic solution to the placement problem. The notion of using a placement algebra to solve the automatic icon placement problem is not new to this thesis.

The placement algebra approach involves specifying location constraints on graphical objects via relational operators. For example, A *is_above* B would constrain graphical object A to be placed somewhere above graphical object B on a layout. Although the absolute locations of objects A and B are not specified in this technique, the relative locations are, and this allows a diagram generator tool to make certain assumptions as to where things are placed providing that the relative placement con-

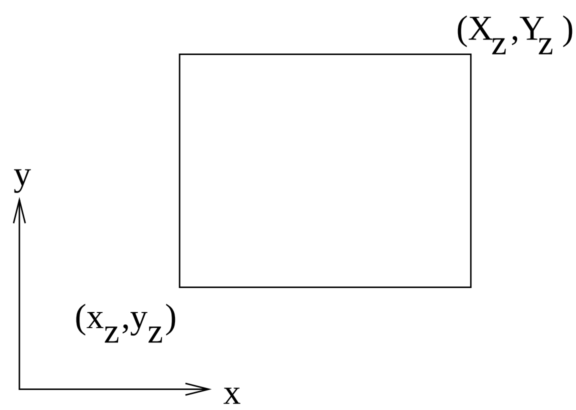


Figure 4.4: Placement Coordinates of Graphical Object z

straints are adhered to. These assumptions involve such decisions as default initial placement and default distance between objects.

The basic algebra for the placement of diagram icons used in DLL is given here. Within the given placement algebra, the $\Rightarrow$ and $\Leftrightarrow$ symbols are used. The $\Rightarrow$ symbol is defined as *logical implication* and the $\Leftrightarrow$ symbol is defined as *logical biconditional*. Assume that a graphical object z is defined by a rectangular bounding box whose lower-left coordinates are (x_z,y_z), and whose upper-right coordinates are (X_z,Y_z), as shown in Figure 4.4. Further, assume that a and b are two such graphical objects, then:

a above b $\Rightarrow$ $y_a > Y_b$
b below a $\Leftrightarrow$ a above b

a left_of b $\Rightarrow$ $X_a < x_b$
b right_of a $\Leftrightarrow$ a left_of b

a in_horiz_line b $\Rightarrow$ $((Y_a - y_a)/2+y_a = (Y_b - y_b)/2+y_b)$
b in_horiz_line a $\Leftrightarrow$ a in_horiz_line b

a in_vert_line b $\Rightarrow$ $((X_a - x_a)/2+x_a = (X_b - x_b)/2+x_b)$
b in_vert_line a $\Leftrightarrow$ a in_vert_line b

a next_to b $\Rightarrow$ there are no icons between a and b

b next_to a $\Leftrightarrow$ a next_to b

a directly_above b $\Rightarrow$ a above b AND a in_vert_line b AND a next_to b

b directly_below a $\Leftrightarrow$ a directly_above b

a directly_left_of b $\Rightarrow$ a left_of b AND a in_horiz_line b AND a next_to b

b directly_right_of a $\Leftrightarrow$ a directly_left_of b

a above_indent b $\Rightarrow$ a above AND a right_of b

b below_indent a $\Leftrightarrow$ a above_indent b

Assume G is a group of icons; the alignment of this group may be defined using any of the relationships here.

a centered_above G $\Rightarrow$ a is located above the geometric midpoint of the group G

a centered_left_of G $\Leftrightarrow$ a is located to the left of the geometric midpoint of the group G

The placement algebra deals with an icon's relationship to the diagram as well as to other icons. If D is the area of the diagram, then:

a contained_within D $\Rightarrow$ $(\mathrm{x}_a > \mathrm{x}_D)$ AND $(\mathrm{X}_A < \mathrm{X}_D)$ AND
$(\mathrm{y}_a > \mathrm{Y}_D)$ AND $(\mathrm{Y}_a < \mathrm{Y}_D)$

a on_exterior_of D $\Rightarrow$ a NOT contained_within D

a center_of D $\Rightarrow$ $((\mathrm{x}_a + \mathrm{X}_a)/2 = (\mathrm{x}_D + \mathrm{X}_D)/2)$
AND $((\mathrm{y}_a = \mathrm{Y}_a)/2 = (\mathrm{y}_D = \mathrm{Y}_D)/2)$

a top_center_of D $\Rightarrow$ $((x_a + X_a)/2 = (x_D + X_D)/2)$ AND $(Y_a = Y_D)$

a left_center_of D $\Rightarrow$ $((y_a = Y_a)/2 = (y_D = Y_D)/2)$ AND $(x_a = x_D)$

a top_left_of D $\Rightarrow$ $(x_a = x_D)$ AND $(Y_a = Y_D)$

a on_border_of D $\Rightarrow$ $(x_a = x_D)$ OR $(Y_a = Y_D)$ OR $(y_a = y_D)$
OR $(X_a = X_D)$

Some special properties of icons are also defined within the placement algebra:

a replicated_in D $\Rightarrow$ a can be replicated in the diagram D
a unique_in D $\Rightarrow$ NOT a replicated_in D

Figure 4.5 further illustrates some of the basic kinds of binary operations that represent the placement algebra defined above. This figure also illustrates that the operators are purely relative and there is no attempt to define exactly how far any object should be from any other object as long as the relative relationship, whatever that may be, is maintained. Absolute placement on a diagram is a decision that is left to the individual layout platform.

Placement algebra addresses the problem of relative icon placement. However, there is more to laying out a diagram than mere placement of individual objects. The idea of diagram orientation must also be addressed. It is possible to categorize diagrams according to their orientation. Orientation refers to whether the diagram is read from top to bottom, left to right or in some other systematic fashion. Software engineering diagrams are typically oriented in three basic ways:

1. Top-down — for example, a structure chart which is illustrated in Figure 4.6(a).
2. Left-right — for example, a left-to-right decision tree as given in Figure 4.6(b).

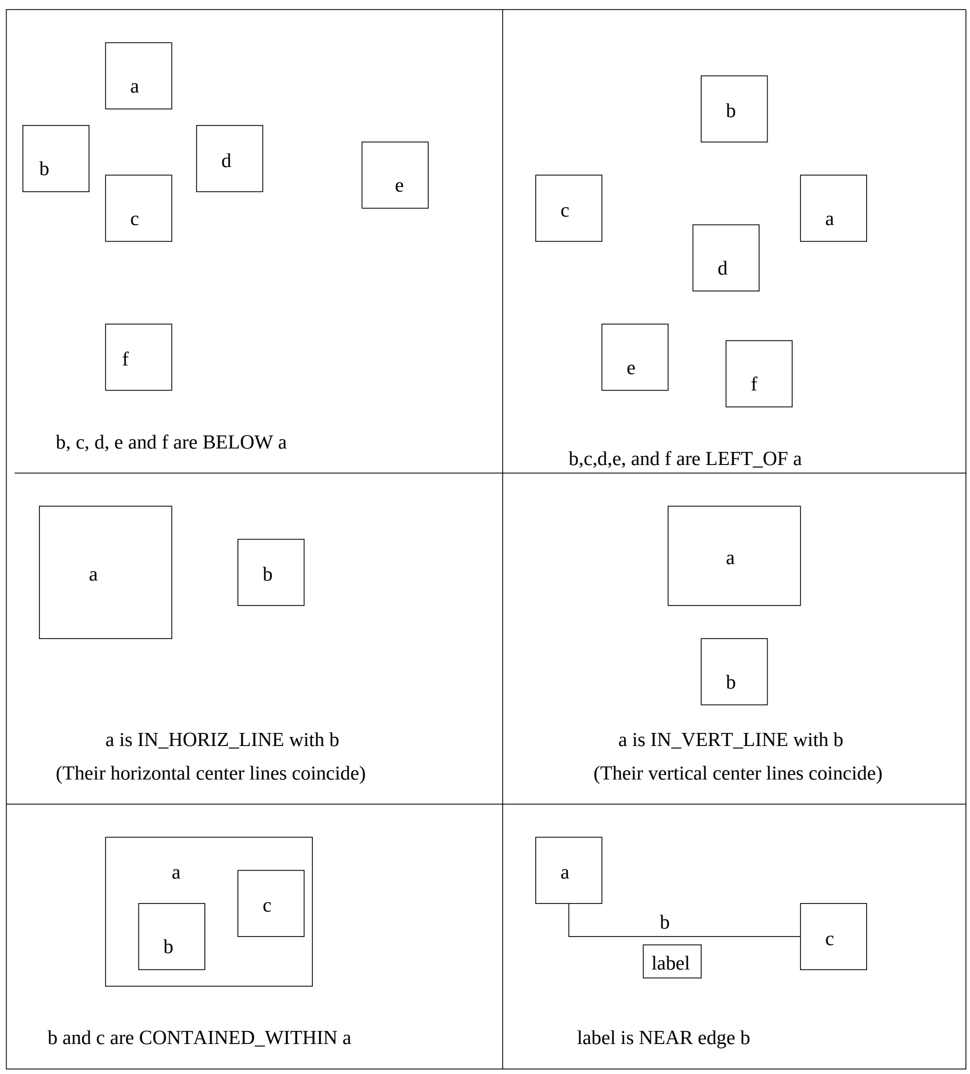


Figure 4.5: Placement Algebraic Relational Operators

3. Not fixed — for example, an entity-relationship diagram as shown in Figure 4.6(c).

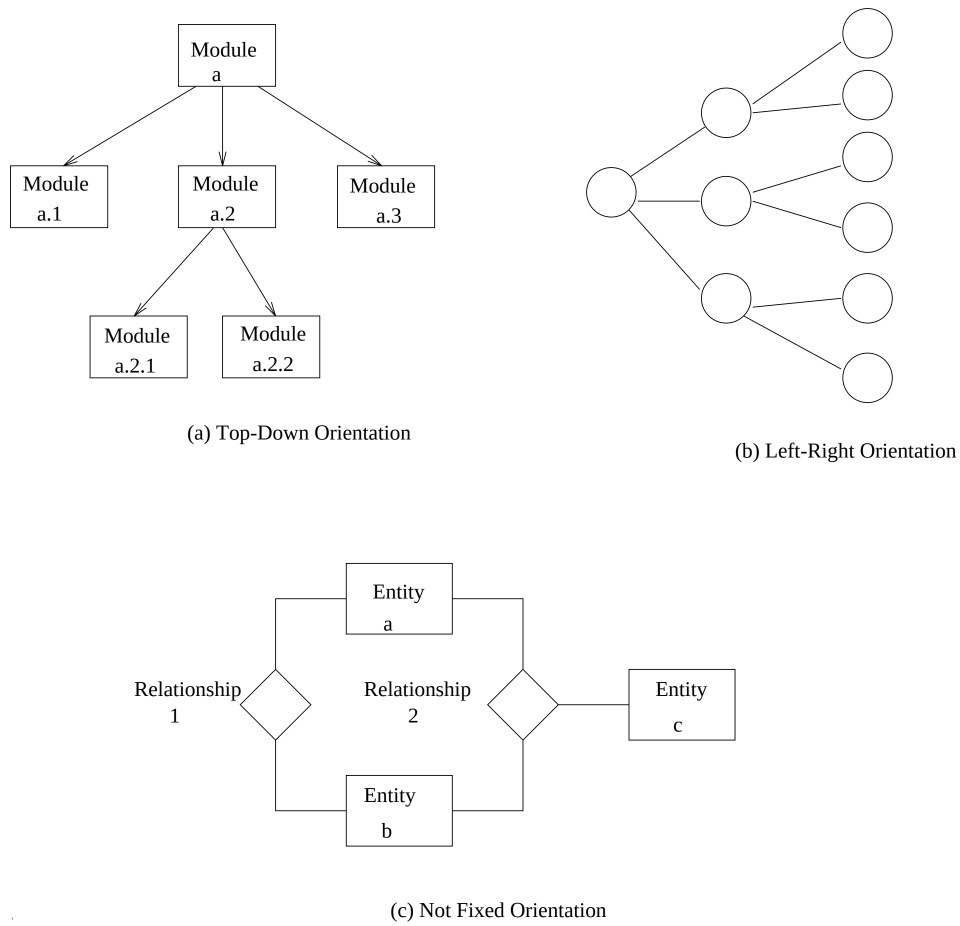


Figure 4.6: Diagram Orientation

The orientation of a diagram can dictate how certain basic relationships are expressed graphically. For example, in a top-down oriented diagram like a structure chart(see Figure 4.6(a)), one node can be considered the "parent" of one or more other nodes. This would imply a location constraint on these nodes in such a way that would force a parent node to be *above* it's children nodes. The children could then have a further location constraint imposed such that they were restricted to be in a horizontal line. Similar location constraints exist for a left-to-right oriented diagram. Thus, once a diagram's orientation is known, it is possible to systematically define placements of icons which are commensurate with the known orientation. As an example, Table 4.1 defines the placement of icons a and b, which represent entities

Table 4.1: Placement Algebra and Diagram Orientation

	not_fixed	top_down	left_right
connected_to(a,b)	left_of(a,b) right_of(a,b) above(a,b) below(a,b)	above(a,b) or possibly below(a,b)	left_of(a,b) or possibly right_of(a,b)
parent_of(a,b)	not applicable except maybe as "explosion"	above(a,b)	left_of(a,b)
sib_of(a,b)	not applicable	in_horiz_line(a,b)	in_vert_line(a,b)

involved in the relationships *connected_to*, *parent_of*, and *sib_of*, according to three different orientations.

Icon groups are also an important aspect of placement algebra. Groups are essential for organizing the placement of icons in some types of diagrams. This is especially true of diagrams that show hierarchies, like structure charts, decomposition diagrams, decision trees, and program structure charts. Wherever there is a hierarchy, there are tiers or levels of superiority and dependence. The members of these tiers should be grouped together and organized appropriately, both with respect to other members within the group, and with respect to the group's parent. Groups can be considered as entities and thus can be placed relative to other entities on the diagram just like individual icons. The next section discusses the motivation for the Diagram Layout Language in which the aforementioned placement algebra is incorporated.

4.4 Motivation for a Diagram Layout Language

In order for the ADL system to generate a diagram, like the one illustrated in Figure 4.3, from the specification database of Section 4.2.2; the ADL system must be able to extract the necessary information from the specification database, specify what graphical objects are to represent this information and then specify how these graphical objects are to be placed relative to one another. The ADL system accomplishes this transition from the textual form to the graphical form via the Diagram Layout Language. The methods engineer encodes all of the necessary diagrammatic

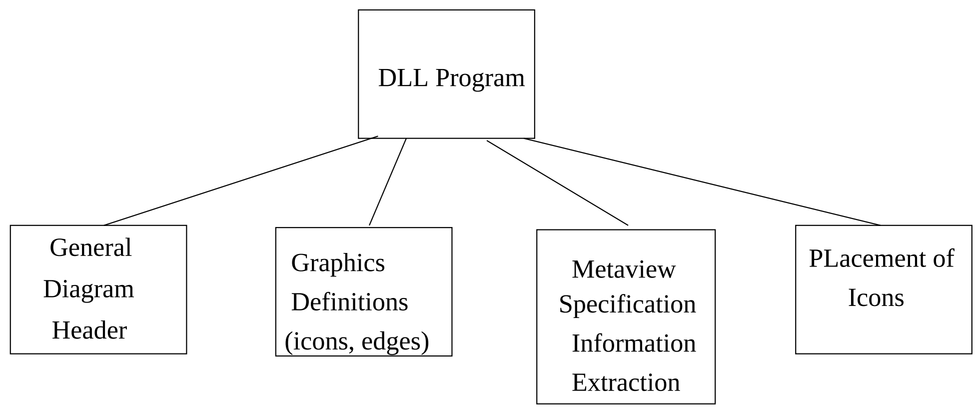


Figure 4.7: Diagram Layout Language High-level Architecture

specifications via DLL constructs.

Figure 4.7 illustrates the main constructs of DLL. In the *General Diagram Header* section, the basic characteristics of the intended diagram are specified such as the general orientation and the overall edge routing standard. In this section of a DLL program, the methods engineer encodes general diagram attributes which are not specific to any one entity or relationship. For example, the overall diagram orientation and edge routing standard could be defined in this section.

After the basic diagram characteristics are defined, the methods engineer can then proceed to define the detailed graphic characteristics of the specific diagrammatic objects. This step is accomplished in the *Graphic Definitions* section of the DLL program. In this section, the detailed graphic attributes of the icons and edges used are specified. For example, it is here that is specified whether or not edges are directed (have arrowheads), and whether or not edges have a label. This section also determines which icons from a predefined *icon library* (refer to Figure 4.1) are to be incorporated into a particular diagram type.

Now that the graphic characteristics of the diagram as a whole and of the individual objects to be laid out have been defined, the methods engineer requires a means of connecting the graphic objects to the conceptual objects defined in the specification database like the one defined in Section 4.2.2. This connection is made in the *Metaview Specification Information Extraction* section of a DLL program. In this section, a methods engineer *maps* the graphic objects defined in the *Graphic Defini-*

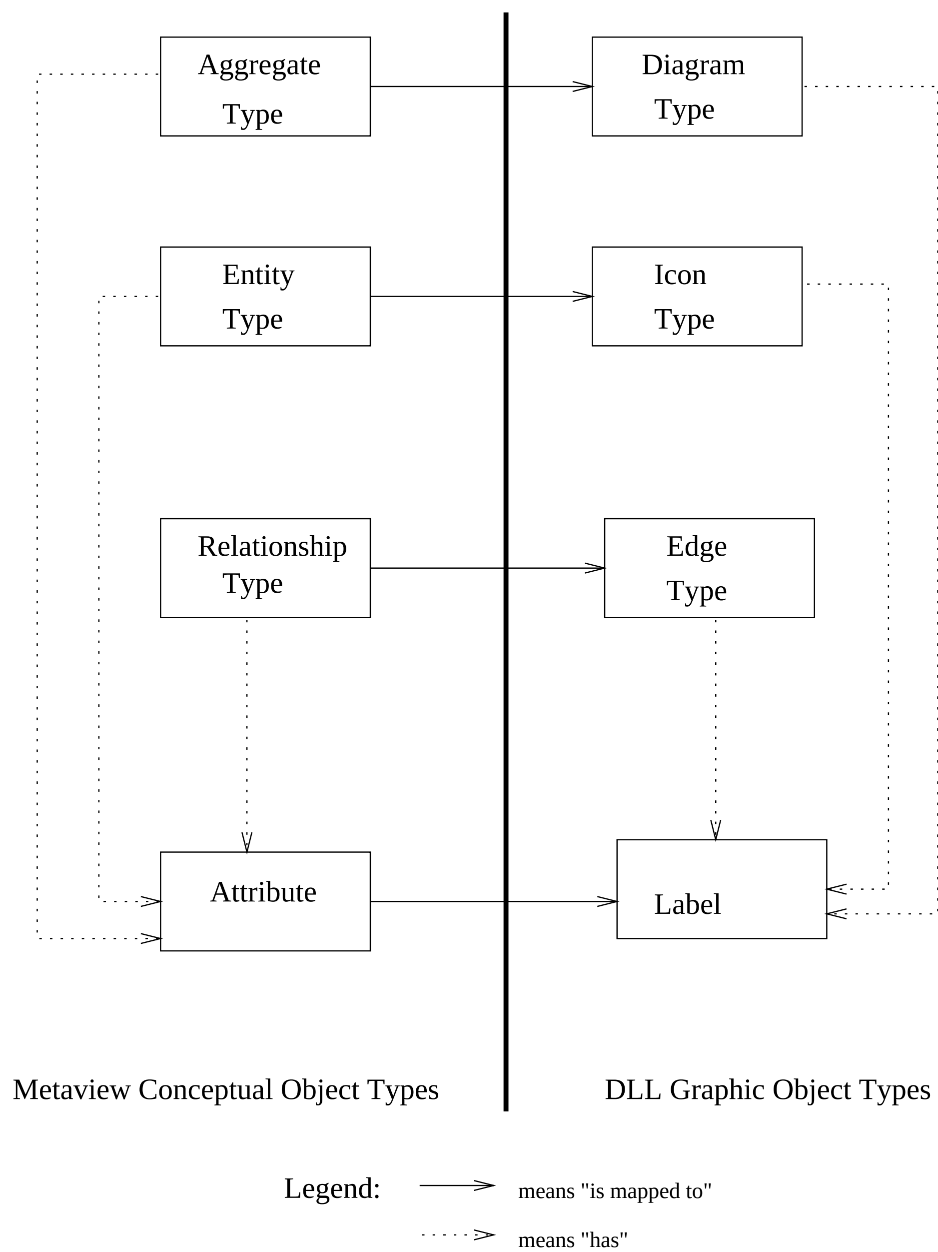


Figure 4.8: Mapping Between Metaview Conceptual and DLL Graphic Objects

tions section of the DLL program to Metaview entity and relationship *types*. Figure 4.8 illustrates this mapping from the conceptual to the graphical as represented in DLL. Note that these graphic object types are mapped to *types* and not to specific instantiations of a type. The reason for this is to keep the diagram definition at the *Metaview Environment* level (refer to Chapter 2). The methods engineer is operating at the environment level. Therefore, the diagram type that is being modeled should be at the environment level, that is, no specific references to instantiated entities or relationships should be made anywhere in the program. This means that each graphic object, defined in the *Graphic Definitions* section, represents a group of instantiated entities and relationships and not necessarily just an instantiated entity or relationship (unless, of course, that within the specification database the entity or relationship type only has one instantiation of itself). For example, the entity type *object_type* has three instantiations of itself, *book*, *fiction*, and *non_fiction*. Whereas, the relationship type *is_supertype* has only one instantiation of itself. Therefore, the DLL defined graphical object which represents the entity type *object_type* will actually represent three icons; the icons will look the same except for, maybe, the label. Whereas, the relationship type *is_supertype*, will have only one edge placed on the diagram. The placement of this edge is performed by the ADL edge router which is described in Section 4.8.

Now that the methods engineer has encoded all of the definitions and extracted the specification information that he/she wishes to lay out diagrammatically, the next logical step is to encode the placement algebraic rules of Section 4.3. This is accomplished via the *Placement of Icons* section of a DLL program. The methods engineer chooses the relationships that he/she wants represented on the diagram and then provides the placement rules for all of the participants of these relationships based on their icon type. For example, it is desired that the relationship type *is_supertype*, and more specifically the relationship instantiation of that type, *is_book*, was intended to be on the diagram. The methods engineer would then encode placement algebraic rules for the corresponding edge type defined in the *Graphical Definitions* section and mapped to the relationship type in the *Metaview Specification Information Extrac-*

tion section and its participating icon types. Encoding placement algebra for an edge and its associated icon types means determining which participating icon types of the desired edge are to be placed relative to the other participating icon types. In the sample method of Section 4.2.1, the relationship type *is_supertype* has the entity types *object_type* and *relationship* (remember this is an entity type here; don't be confused by its name) as participants. Therefore, the methods engineer could encode the fact that he/she wishes that for any edge, mapped to the relationship type *is_supertype*, with those entity types as participants, always place any instance of the entity type *object_type ABOVE* any instance of the entity type *relationship.* Therefore, in the sample specification database of Section 4.2.2, the icon representing the object *book* would be placed *ABOVE* the icon representing the object *relationship0.* All placement definitions defined in the *Placement of Icons* section of a DLL program would be defined in a similar fashion.

The Diagram Layout Language described in this thesis is intended to allow the methods engineer to have control over the diagram layout process. The language is a vehicle for modeling a diagram layout. The methods engineer's preferences and expertise are stated in a DLL program. The DLL compiler merely translates the methods engineer's intent, which is encoded in the program, into diagram modeling tables which are then used to transform a specification database into a diagram like the one shown in Figure 4.3.

A benefit of the approach taken in this thesis is the ease in which a methods engineer can initially model a diagram. This model of the diagram (the DLL program) can be subsequently modified any time to handle changes in the system developer's and the methods engineer's views of the way in which the icons, edges, and associated graphical properties should appear on the diagram.

4.5 Introduction to the Diagram Layout Language (DLL)

The Diagram Layout Language represents the first component of the ADL System. DLL, which relies heavily on the placement algebraic approach presented in Section 4.3, is the starting point for a methods engineer to model a specific family of diagrams. In this section, DLL will be explained at a high level so that the reader can get an overall grasp of DLL before defining its individual pieces. A sample DLL program, along with a corresponding Metaview method definition and a specification database instantiation of that method, has been presented in Section 4.2.

4.5.1 Language Components

A DLL program has four main sections. These are:

1. General Diagram Header — This section is the introductory header to the program. The statements in this section are used to specify general aspects of the diagram type being modeled. These aspects are program name, diagram name, diagram orientation, and routing standard.

2. Graphic Definitions — This section contains statements which are used to capture the graphic details of the icon types to be used and the edge types to be routed between the icon types. The statements determine what icon types are to be used and what edge type to use for certain relationships.

3. The Metaview Connection — This section is the link between the ADL tool and the Metaview metacase system. The statements in this section determine which entity types and relationship types in a specific method specification are to be used in the layout of the diagram being modeled. A methods engineer restricts the entity types and relationship types chosen to be components of one specific aggregate type because aggregate types are mapped to diagram types.

```
PROGRAM: wmrt;
DIAGRAM_NAME: data_schema;
ORIENTATION: TOP_DOWN;
STANDARD: GRID;
```

Figure 4.9: Sample DLL Program Segment: General Diagram Header

4. Placement of Icons — This section contains statements which are central to a DLL program. These statements are placement algebra specific. Any relationship which has been accessed via the above Metaview connection statements should be represented here. Essentially, the methods engineer codes placement algebraic statements for each combination of participants for each Metaview relationship type represented.

4.5.2 A Sample DLL Program — Input to the DLL Compiler

The following code samples are segments of a DLL modeling of an entity-relationship diagram. Most aspects of the language are illustrated with these samples.

General Diagram Header: A typical header section for a DLL program is illustrated in Figure 4.9. The program's and diagram's names have been identified, along with the intended orientation and routing standard. In this code, the program name is *wmrt* (Ward and Mellor Real-Time) which indicates the Ward and Mellor Real-Time method. The diagram name is *data_schema* which indicates the Metaview aggregate type (see Figure 4.2) within the WMRT method definition. The orientation of the diagram type is specified as *top_down*, which indicates that the diagram *flows* from the top down. The routing standard is stated as *grid*, which indicates that only 90 degree bends are allowed on the routed edges of the diagram.

Graphical Definitions: The graphics definitions section of a DLL program specifies what icon types are to be used from the icon library (refer to Figure 4.1). This

```
ICON_TYPES: {LABELLED ( entity_icon, relationship_icon) };
ARC_TYPES: {NON_LABELLED NON_DIRECTED ( association_uni_flow,
                    is_supertype_uni_flow) };
```

Figure 4.10: Sample DLL Program Segment: Graphics Definitions

section also allows the methods engineer to specify the graphical characteristics of the icon and edge types used in the diagram. In Figure 4.10, the graphic definitions for the icons and edges are specified. As the figure shows, all icon types are labeled. All edge types are not labeled and have no arrowheads.

Metaview Connection: The Metaview connection section of a DLL program is responsible for mapping the Metaview conceptual entity and relationship types to the DLL icon and edge types, respectively, (see Figure 4.11). This section also allows the methods engineer to specify an alternate label to the default label. The default label for icon types is the implicit *name* attribute of the corresponding entity type defined in the method definition. The default label for relationship types is not its *name* attribute, because relationship types don't possess an implicit *name* attribute. Therefore, the default label for an edge is the concatenation of the *name* attributes for the participating entity types that are not the *sender* nor the *receiver*. The notion of sender and receiver participants is an important concept in the design of the Diagram Layout Language. In a relationship type, there could be many roles with many participating entity types for each role. In a DLL program, the two main roles are *sender* and *receiver* because DLL can currently only model binary relationships. This, however, does not imply that higher arity relationship types can not be represented on a diagram. That would restrict DLL too severely. Therefore a means of specifiying the sender and receiver role participants is provided. The methods engineer specifies the participating entity types of a relationship type to be the sender and receiver participants for the binary edge types routed on the diagram.

Figure 4.11 illustrates the syntax of a typical Metaview connection section of a DLL program. The DLL code maps the DLL icon types and edge types to the cor-

```
MAP_REL
    association_uni_flow TO association;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    is_supertypeuni_flow TO is_supertype;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    binds_uni_flow TO binds;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_ENTITY entity_icon TO object_type;
MAP_ENTITY
    relationship_icon TO relationship;
    LABEL IS {relationship_name}
```

Figure 4.11: Sample DLL Program Segment: Metaview Connection

responding Metaview entity and relationship types, respectively. As well as mapping DLL objects to Metaview objects, the participants, and, optionally, the labels for each object are specified. For example, the DLL edge identifier *association_uni_flow* is mapped to the corresponding Metaview relationship type *association*. The code also maps the participants in roles one and two of the Metaview *association* relationship type to the sending and receiving roles, respectively, of the *association_uni_flow* edge type. Also, the DLL icon identifier *relationship_icon* is mapped to the Metaview entity type *relationship*. The label of this icon is the entity type *relationship's relationship_name* attribute. The *LABEL IS* clause is optional for both the *MAP_REL* and *MAP_ENTITY* statements.

Placement of Icons: The placement of icons section represents the main component of a DLL program. This code leads to the generation of the necessary placement constraints for all of the defined icon types in Figure 4.11. Essentially, for each edge type defined in Figure 4.11, a *RELATIONSHIP* block exists in which the methods engineer can handle all desired permutations of the possible participants for each edge. Figure 4.12 illustrates a typical icon placement section of a DLL program. The *ASSOCIATE* statements map local DLL identifiers to icon types defined in Figures 4.10 and 4.11. The local identifiers are statically scoped.

```
RULE_BLOCK
    ASSOCIATE ent: entity_icon;
    ASSOCIATE rel: relationship_icon;
    RELATIONSHIP is_supertype_uni_flow( _ , _ )
        CASE ent , rel:
            ABOVE( ent , rel ) AND LEFT_OF( ent , rel );
        ENDCASE
    ENDRELATIONSHIP
ENDBLOCK
END //  END terminates a DLL program
```

Figure 4.12: Sample DLL Program Segment: Placement of Icons

For each possible combination of sending and receiving participants in a relationship type, a *CASE* statement is coded. For example, in Figure 4.12, the relationship type *is_supertype*, which is mapped to the edge type *is_supertype_uni_flow* has only one combination of participating entity types, therefore, there is only one *CASE* statement coded to handle this situation. In this case, all instances of the *is_supertype* relationship type would have all of the participating *entity_icon* instances placed *ABOVE* and to the *LEFT_OF* all the participating *relationship_icon* instances. The fact that the local identifier *ent* appears ahead of the identifier *rel* is significant in the sense that the order of these identifiers dictates the order that the participating entity types must be for the current relationship type in the current *RELATIONSHIP* block.

4.6 Definition of the Diagram Layout Language

The following sections define the detailed design grammar of the Diagram Layout Language in terms of the main components of the language which were outlined in Section 4.4.

4.6.1 Extended BNF

Before formally defining DLL, the notation used to define the formal design grammar is explained.

A context-free grammar (CFG), G, is a 4-tuple:

$$G = (V_N, V_T, S, \Phi)$$

where V_N is the set of non-terminal symbols, V_T is the set of terminal symbols, S is the start symbol, and Φ is a set of productions of the form $\alpha \rightarrow \beta$, where $|\alpha| \leq |\beta|$ and $\alpha \in V_N$. For example, consider the following grammar: $G = (\{S, C\}, \{a, b\}, S, \Phi)$ and Φ is the following set of productions

$S \rightarrow aCa$

$C \rightarrow aCa$

$C \rightarrow b$

A derivation for a^2ba^2 consists of the following steps:

S $\Longrightarrow$ aCa $\Longrightarrow$ aaCaa $\Longrightarrow$ aabaa

where "$\Longrightarrow$" denotes the directly produces relationship.

BNF (Backus Naur Form) has been used extensively in the formal definition of many programming languages. The two metasymbols used in BNF are:

- ::= — This metasymbol represents "is defined as" and replaces the symbol $\rightarrow$ in the grammar notation.

- | — This metasymbol represents "or" and is used to separate different right-hand sides of productions corresponding to the same left-hand side.

Lowercase words are non-terminal symbols (V_N) of the grammar, and uppercase words and punctuation symbols are terminal symbols (V_T) of the grammar. Therefore, the grammar, G, would then be represented in BNF as: $G = (\{s, c\}, \{A, B\}, s, \Phi)$ and Φ is the following set of productions s ::= AcA

c ::= AcA | B

The grammar, G, has not changed in the sense that it still produces the same language.

BNF is now extended to include the following metasymbols:

- $[\sigma]$ — These metasymbols, [and], denote optional elements. The element, σ, enclosed within the brackets could be produced zero or one time.
- $\{\sigma\}_a^b$ — These metasymbols, { and $\}_a^b$, denote iteration of the element σ. It means that σ could be produced a minimum of a times to a maximum of b times. If a is omitted then zero is assumed for the lower limit. If b is omitted then ∞ is assumed. Therefore, $\{\sigma\}^1$ is equivalent to $[\sigma]$.
- $(\sigma_1|\sigma_1|\cdots|\sigma_n)$ — The metasymbols, (and), are used to denote alternative groupings within the right-hand side of a production.

Whenever a symbol is used both as a terminal and a BNF metasymbol, it will be enclosed in single quotes to denote that it is a symbol of the grammar and not of the BNF. For example, if a left parenthesis is part of the grammar then it will be represented as '(' to distinguish from the extended BNF metasymbol (.

The formal DLL design grammar has a different appearance than the actual input grammar to ATS[3] as presented in Appendix A. The reason for this difference in appearance is due to the fact that the ATS input grammar is required to be an LL(1) grammar. The DLL design grammar is not required to be LL(1). However, the two grammars only appear to be different as they both specify the same language.

4.6.2 Lexical Conventions

There are three classes of tokens: identifiers, keywords, and operators/separators. Blanks, tabs, newlines, and comments (collectively, "white spaces") are ignored except when used to separate tokens. DLL is case-sensitive.

4.6.2.1 Comments

Comments in a DLL program are denoted by // and carriage return for single line comments.

4.6.2.2 Reserved Words, Identifiers and Operators/Separators

Table 4.2 lists the DLL keywords. The DLL keywords are reserved and may not be used as identifiers in a DLL program.

An identifier in DLL is a sequence of letters, digits, and special characters (underscores and hyphens). The first character of any identifier should be a letter.

The list of separators and operators is presented in Table 4.3. There is some overlap between Table 4.2 and Table 4.3. This is because some keywords are also operators.

4.6.3 General Diagram Header

4.6.3.1 Grammar

The BNF for the general diagram header section of a DLL program follows. This section of the grammar defines the general attributes of the diagram type being modeled.

program ::= program_header diagram_name_declaration orientation_declaration
  standard_declaration icon_type_declaration arc_type_declaration
  mapping_declaration placement_rules END

program_header ::= PROGRAM : identifier ;

identifier ::= letter { letter | _ | - | digit }

letter ::= lowercase_letter | uppercase_letter

diagram_name_declaration ::= [DIAGRAM_TYPE_NAME : identifier ;]

orientation_declaration ::= [ORIENTATION : orientation ;]

orientation ::= TOP_DOWN | LEFT_RIGHT | NOT_FIXED

standard_declaration ::= [STANDARD : standard ;]

Table 4.2: List of Keywords in DLL

ABOVE	ABOVE_INDENT	AND
ARC_TYPES	ASSOCIATE	BELOW
BELOW_INDENT	BI_DIRECTIONAL	CASE
CENTERED_ABOVE	CENTERED_BELOW	CENTERED_LEFT_OF
CENTERED_RIGHT_OF	CENTER_OF	CENTER_OF
CONFLICT	CONTAINED_IN	CURVED
DIAGONAL	DIAGRAM_TYPE_NAME	DIRECTLY_ABOVE
DIRECTLY_BELOW	DIRECTLY_LEFT_OF	DIRECTLY_RIGHT_OF
END	ENDBLOCK	ENDCASE
ENDPOINT1	ENDPOINT2	ENDRELATIONSHIP
EXTERIOR_TO	GRID	GROUP
ICON_TYPES	IN_HORIZ_LINE	IN_VERT_LINE
IS	LABEL	LABELED
LEFT_CENTER_OF	LEFT_OF	LEFT_OF
LEFT_RIGHT	MAP_ENTITY	MAP_REL
MIXED	NEXT_TO	NON_DIRECTED
NON_LABELED	NOT_FIXED	ON_BORDER_OF
ORIENTATION	PRIORITY_IS	PROGRAM
RECEIVER	RELATIONSHIP	REPLICATED_IN
RESOLUTION_MAPPING	RIGHT_OF	ROLE
RULE_BLOCK	SENDER	STANDARD
TO	TOP_CENTER_OF	TOP_DOWN
TOP_LEFT_OF	UNIQUE_IN	UNI_DIRECTIONAL

Table 4.3: List of Separators and Operators in DLL

(	)	;
{	}	_
=	,	|
:	ABOVE	ABOVE_INDENT
BELOW	BELOW_INDENT	CENTERED_ABOVE
CENTERED_BELOW	CENTERED_LEFT_OF	CENTERED_RIGHT_OF
CENTER_OF	CONTAINED_IN	DIRECTLY_ABOVE
DIRECTLY_BELOW	DIRECTLY_LEFT_OF	DIRECTLY_RIGHT_OF
EXTERIOR_TO	IN_HORIZ_LINE	IN_VERT_LINE
LEFT_CENTER_OF	LEFT_OF	ON_BORDER_OF
REPLICATED_IN	RIGHT_OF	TOP_CENTER_OF
TOP_LEFT_OF	UNIQUE_IN	

standard ::= GRID | DIAGONAL | CURVED | MIXED

4.6.3.2 Semantics

The DLL program begins with an introductory section that declares the diagram's type, orientation, and standard, as well as icon types and arc types to be used in the type of diagram being modeled.

The diagram type name refers to the name of the desired aggregate type within the intended Metaview method specification. The diagram's orientation specifies the overall flow of the diagram. The standard declaration refers to the type of routing used by the ADL system. Currently only GRID routing is supported. The types of routing standards are:

- grid — This type of routing only allows edges to bend at right angles.
- diagonal — This type of routing allows edges, which connect icons, to be at other than right angles to the main X-Y axes of the plane in which the diagram is being drawn.
- curved — This type of routing standard allows edges to form arcs rather than just straight lines.
- mixed — This type of routing standard allows all of the above types on a single diagram. This is the most flexible, and thus the most difficult to implement.

Figure 4.13 illustrates the four routing standards.

4.6.3.3 Examples

The following DLL code fragment is from a Ward and Mellor Real-Time diagram modeling DLL program:

```
PROGRAM: wmrt;
DIAGRAM_TYPE_NAME: data_schema;
```

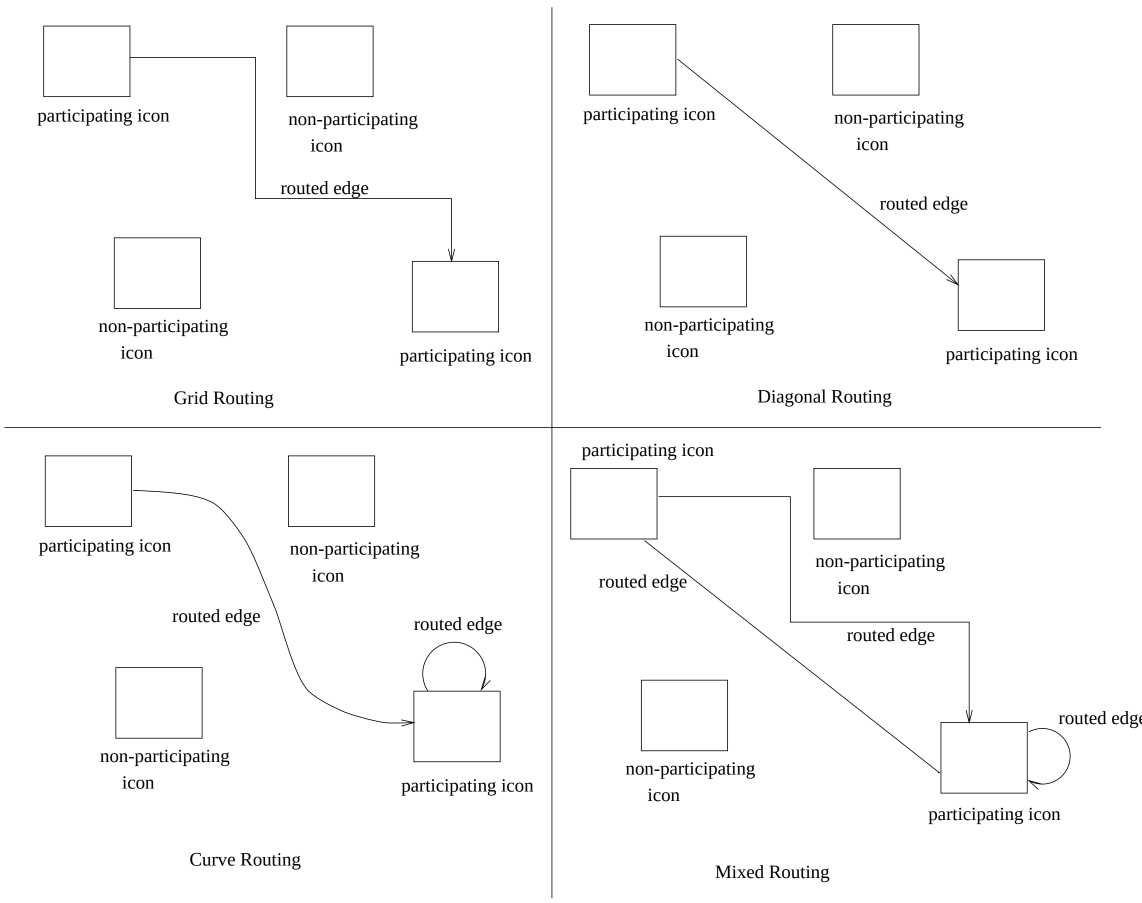


Figure 4.13: Software Diagram Routing Standards

```
ORIENTATION: TOP_DOWN;
STANDARD: GRID;
```

The next DLL code fragment is from modeling a Data Flow Diagram:

```
PROGRAM: dfd;
DIAGRAM_TYPE_NAME: context_diagram;
ORIENTATION: LEFT_RIGHT;
STANDARD: GRID;
```

4.6.4 Graphical Definitions

The BNF for the graphic definitions section of a DLL program follows. This grammar fragment defines specific graphic attributes of the diagram's icons and edges. For example, this section specifies whether or not certain icon and edge types are labeled or not. This section also specifies whether or not edges are directed.

4.6.4.1 Grammar

There are two types of clauses in this section. One clause defines the icons and their attributes, and the other clause defines the edges and their attributes.

icon_type_declaration ::= [ICON_TYPES : '{' icon_type_list
  {, icon_type_list } '}' ;]
icon_type_list ::= { label_status '(' identifier {, identifier} ')'}$_1$
arc_type_declaration ::= [ARC_TYPES : '{' arc_type_list {, arc_type_list } '}' ;]
arc_type_list ::= { label_status direction '(' arc_type {, arc_type} ')' }$_1$];
label_status ::= LABELED | NON_LABELED
direction ::= UNI_DIRECTIONAL | BI_DIRECTIONAL | NON_DIRECTED
arc_type ::= identifier

The clauses defined in this section allow the methods engineer to easily define common attributes for some or all of the icon and edge types by specifying the attributes first and then listing the icons or edges afterwards which have these attributes. For example, if all icons are labeled, then only one *ICON_TYPES* clause would be required. Edges are handled similarly.

4.6.4.2 Semantics

The *icon_type_declaration* lists the types of icons which could be used in a specific diagram type. This list refers to icon definitions listed in the icon library referred to in Figure 4.1. If an icon type is mentioned here that is not in the library, an appropriate error message is displayed.

The *arc_type_declaration* lists the types of arcs which will be found in a specific diagram type. This declaration explicitly declares whether the intended type of arcs will be labeled or not as well as whether or not arrowheads should be attached to the arc when it has been routed. Currently, ADL supports only binary relationships, thus, the arc type mentioned is not accompanied by a corresponding entity type list of participants to determine arity of the relationship.

The presence of the colon, in both *icon_type_declaration* and *arc_type_declaration*, reflect the personal preferences of the designer.

4.6.4.3 Examples

The first example is taken from the modeling of a Ward and Mellor Real-Time diagram type:

```
ICON_TYPES: {LABELED (entity_icon, relationship_icon)};
ARC_TYPES: {NON_LABELED NON_DIRECTED (association_uni_flow,
            is_supertype_uni_flow, binds_uni_flow) };
```

This example specifies that all icons are labeled. All edges, however, are not labeled, nor do they possess arrowheads (direction). This sample code is likely what would be specified for the Entity-Relationship method defined in Figure 4.2 of Section 4.2.

4.6.5 Metaview Connection

The Metaview connection section of a DLL program defines the mapping between the Metaview conceptual types and the DLL graphical types. Refer to Figure 4.8 to see the explicit mapping. This section also allows the methods engineer to specify the label sources for the edge and icon types to be used in the diagram. For example, an entity type's default label is its *name* attribute, however, the methods engineer might prefer to specify another of the entity type's attributes to be the label. Similarly, a methods engineer might prefer to change the default label of an edge type.

4.6.5.1 Grammar

There are two types of clauses in this section. One clause maps icon types to entity types, and the other clause maps edge types to relationship types. Both clauses possess a mandatory portion for the mapping and an optional portion for changing the default label.

```
mapping_declaration ::= {mapping}
mapping ::= MAP_ENTITY map_entity_tail ; | MAP_REL map_rel_tail ;
map_entity_tail ::= identifier TO identifier [LABEL IS '{' identifier '}' ]
map_rel_tail ::= identifier TO identifier
    SENDER IS ROLE NUMBER
    RECEIVER IS ROLE NUMBER
    [LABEL IS '{' identifier '}' ]
```

The first identifier in *map_entity_tail* and *map_rel_tail* correspond to the icon type and edge type name, respectively. The second identifier, in the same productions, are the corresponding entity type and relationship type names defined in the method.

The syntax of the *MAP_REL* clause is somewhat verbose because it was felt that less confusion would arise.

4.6.5.2 Semantics

The *mapping_declaration* clauses are the means by which Metaview/EARA entity types are mapped to icon types and Metaview/EARA relationship types are mapped to arc (or edge) types. The *MAP_ENTITY* clause maps icon types found in the icon library of the ADL system to Metaview entity types. These DLL icon types are used later in the placement rules section of the grammar. The one option that the *MAP_ENTITY* clause has is the ability to redefine the icon's label via the *LABEL IS* clause. If the latter clause is present then the entity type's attribute referred to within the braces is used. If the *LABEL IS* clause is omitted then the default label is used to label the icon. The default label for an icon is the entity type's *name* attribute.

The *MAP_REL* clause maps DLL arc types to Metaview relationship types. These DLL arc types are used later in the placement rules section of the grammar. Currently the ADL system can only handle binary arc types whereas Metaview's relationship types allow any finite arity. DLL uses the *SENDER IS ROLE* and *RECEIVER IS ROLE* constructs to allow the diagram developer to choose which two roles in the relationship type should be mapped to the *sender* and *receiver* roles in DLL. This is a reasonable way of handling the multi-arity problem because Metaview environment definers tend to define relationships in such a way as to represent the thing flowing from one entity to another as a participating entity type or participating entity types.

The *MAP_REL* clause has the optional *LABEL IS* clause as well. The default for this statement, however, is not the name attribute of the Metaview relationship type. This is because there is no implicit relationship type *name* attribute. Thus, the default was chosen to be the concatenation of the *name* attributes of the participating entity types in the relationship type which are not fulfilling either the *sender* or *receiver* role. This heuristic is crude, but effective.

4.6.5.3 Example

The following DLL fragment is from a Ward and Mellor Real-Time (WMRT) diagram modeling program. The corresponding Metaview method definition can be found in Figure 4.2. This method corresponds to the *Data Schema* aggregate type within the WMRT method and closely resembles the Entity Relationship model developed by Chen [4].

```
MAP_REL
    association_uni_flow TO association;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    is_supertype_uni_flow TO is_supertype;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    binds_uni_flow TO binds;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_ENTITY entity_icon TO object_type;
MAP_ENTITY
    relationship_icon TO relationship;
    LABEL IS { relationship_name }
```

This code fragment illustrates the fact that there are three Metaview relationship types being mapped to three DLL edge types and the primary entity type participants for all three relationship types are the first and second participating entity types in the formal definition of the relationship types. In Figure 4.2 the three relationship types, *association, is_supertype*, and *binds*, are clearly defined along with their possible entity type participants. For example, in the DLL fragment, the relationship type *association* has two entity types as possible participants to the two given roles. The *MAP_REL* statement for the edge type *association_uni_flow*, which is mapped to the relationship type *association*, states that when an instance of this edge type is routed on a specific diagram type instance, then the edge is to be routed *from* the entity type *object_type*, the *sender* role participant, to the entity type *relationship*, the *receiving* role participant.

The code fragment also depicts the fact that there are two entity types being mapped to their corresponding DLL icon types. These entity types are also formally defined in the Metaview ER method definition in Figure 4.2. The *MAP_ENTITY* statement for the *relationship* entity type has made use of the optional *LABEL IS* clause to redefine the corresponding icon types label to be the attribute *relationship_name* instead of the default attribute *name*.

4.6.6 Placement of Icons

4.6.6.1 Icon Placement Grammar

The BNF for the placement of icons section of a DLL program will now be given. This section of the grammar defines the location constraints via the placement algebra of Section 4.3.

The main segments of the the icon placement grammar are the *rule block*, *local identifiers*, *special placement*, *placement scope*, *placement algebra*, and *conflict resolution*. Within a *rule block*, local identifers are declared which associate identifiers to an icon types. Any *special placement* rules which place icons relative to the diagram as a whole are then encoded. The *placement scope* for placement algebra statements is then encoded. Establishing a scope for the placement algebra involves specifying an edge type via the *RELATIONSHIP* clause, and then picking a particular combination of sender and receiver participants via the *CASE* clause. Within a *CASE* clause, the placement algebra is specified for the sender and receiver icon types. Once all of the placement algebra has been specified, any direct placement conflicts that might arise are focused on, and resolved with the *conflict resolution* segment.

Rule Block:

This grammar fragment defines the basic rule block structure of DLL. A rule block possesses local identifiers specified at the beginning of the block. The rule block can then have special placement rules, relative placement rules, and conflict resolution rules, in that order.

placement_rules ::= {rule_block}

rule_block ::= RULE_BLOCK {ident_with_icon}$_1$ {special_placement} {rule}$_1$
{conflict_resolution} ENDBLOCK

The rule block specifies all of the placement rules for a given diagram type.

Local Identifiers:

The first identifier mentioned in the *ident_with_icon* rule is the local scope variable which gets linked to the second identifier. The first identifier is called an *icon denoter*. The second identifier refers to the icon type which was mapped to a specific Metaview entity type(s) in a preceding *MAP_ENTITY* statement.

ident_with_icon ::= ASSOCIATE identifier : identifier ;

The preceding grammar specifies how local identifiers are declared.

Special Placement:

The special placement section of a rule block has syntax which specifies placement of icons within a group (*group placement*). It also has syntax to specify placement of icons relative to the diagram. There is also syntax which allows specification of icon *placement properties*.

special_placement ::= group_placement '(' identifier ')' ;
| icon_diagram_placement '(' identifier ')' ;
| placement_property '(' identifier ')' ;

group_placement ::= IN_HORIZ_LINE | IN_VERT_LINE

icon_diagram_placement ::= CONTAINED_IN | EXTERIOR_TO | CENTER_OF |
TOP_CENTER_OF | LEFT_CENTER_OF | TOP_LEFT_OF |
ON_BORDER_OF

placement_property ::= REPLICATED_IN | UNIQUE_IN

The given grammar outlines the means by which icon placement is specified which is not relative to other icons.

Placement Scope:

This grammar fragment specifies an edge type name (*RELATIONSHIP clause*) to be used as the scope for corresponding *CASE* clauses. Each *RELATIONSHIP clause* can have many *CASE* clauses. It also specifies the definition of a *grouping* of icons to be used within the scope of the *RELATIONSHIP* clause.

rule ::= case_statement | grouping ;

case_statement ::= RELATIONSHIP arc_type2 {case} ENDRELATIONSHIP

arc_type2 ::= identifier '(' (identifier , _ | identifier , identifier | _ , _
  | _ , identifier) ')'

case ::= CASE (identifier , _ | identifier , identifier | _ , _ | _ ,identifier)
  : placement_statement_list ENDCASE

grouping ::= GROUP identifier : identifier {, identifier}

The preceding grammar specifies the scope for the clauses involving the placement algebra.

Placement Algebra:

The placement algebra statements are encoded within each *CASE* clause. Each *placement_statement_list* is a conjunction of placement algebra operators (*icon_icon_placement*) whose operands correspond to the icon types specified in the parent *CASE* clause.

placement_statement_list ::= simple_placement_expr ; {simple_placement_expr ;}

simple_placement_expr ::= placement_term

placement_term ::= placement_factor {AND placement_factor}

placement_factor ::= placement_definition | grouping

placement_definition ::= icon_icon_or_group_placement icon_icon_or_group_placement ::= icon_icon_placement [participants]

| group_placement '(' identifier ')'

participants ::= '(' (identifier | _) , (identifier | _) ')'

icon_icon_placement ::= ABOVE | BELOW | LEFT_OF | RIGHT_OF |

ABOVE_INDENT | BELOW_INDENT | DIRECTLY_ABOVE |

DIRECTLY_BELOW | DIRECTLY_LEFT_OF | DIRECTLY_RIGHT_OF |

CENTERED_ABOVE |CENTERED_BELOW | CENTERED_LEFT_OF |

CENTERED_RIGHT_OF | NEXT_TO

The preceding grammar specifies the placement algebra of the Diagram Layout Language.

Conflict Resolution:

The grammar for the resolution of direct placement conflicts specifies the possible edge types that might have conflicting placement constraints for their participating sending and receiving entity types. The grammar gives the methods engineer the capability of focusing on these types of conflicts and choosing which placement constraints are to be relaxed so as to avoid the conflict(s).

The *conflict_resolution* construct identifies the relationships, via the *CONFLICT* construct, in which a possible conflict could arise and then notes where, specifically, the conflict could be and gives priority to one of the relationship's participants, via the *RESOLUTION_MAPPING* construct.

conflict_resolution ::= CONFLICT : '{' edge_ident_list '}'

RESOLUTION_MAPPING : '{'res_mapping_statement'}'

edge_ident_list ::= identifier '(' icon_types ,icon_types ')'

{,identifier '(' icon_types ,icon_types ')'}$_1$;

icon_types ::= identifier {'|' identifier}

res_mapping_statement ::= simple_mapping_expr : PRIORITY_IS simple_res_expr ;

simple_mapping_expr ::= mapping_term

mapping_term ::= mapping_factor {AND mapping_factor} mapping_factor ::= identifier '(' role ')' = identifier '(' role ')'

role ::= SENDER | RECEIVER

simple_res_expr ::= res_term

res_term ::= res_factor {AND res_factor}

res_factor ::= identifier '(' role ')'

The preceding grammar specifies a set of edge types which might have placement constraint conflicts associated with them. The grammar then specifies what the conflict condition is for pairs of edge types, and what the priority edge type and role should be to resolve the conflict.

4.6.6.2 Semantics

The preceding grammar grammar fragment formally specifies placement algebra. This section describes how the placement algebra concept is incorporated within the DLL design framework.

The main body of a DLL program consists of *rule block* segments which are composed of *associate* clauses, *special placement* clauses, *rule* clauses, and *conflict resolution* clauses. The *associate* clauses are used to define local scope identifiers to be associated with the global icon definitions already defined. The *special placement* clauses are intended to be used for the placement of icons relative to the diagram. The *rule* clauses incorporate the placement algebra. The *conflict resolution* clause allows the developer to isolate possible placement conflicts and resolve them according to the developer's intentions.

Within a rule block, the association between local icon type identifiers and the global icon types, declared in the graphics definitions section, is specified in the *local identifiers* segment within the *ASSOCIATION* clause. Here all icon denoters to be used within this rule block are declared to be of a certain icon type. This local association will hold throughout the rule block, but may be reassigned within a different rule block. By using such an association, we can eliminate any further references to global icon types within the block. The rule block is the scope of the icon identifiers.

The *special placement* segment refers to the placement of icons or groups relative to the whole diagram and not merely other icons or groups within the diagram. Special placement also refers to specifying *placement properties* of specific icon types. For example, some icon types may be duplicated on a diagram. Data Flow Diagrams have external entity icons which may be replicated. There is also a need to specify the relative placement of icons within a group. For example, in hierarchical diagrams, a group of sibling icons, which are all below their parent icon, would probably appear in a horizontal line for good aesthetics.

Two basic *rule* constructs may be used within the *placement scope* segment. The *case statement* is the basic construct which operates on a per edge/relationship type basis. This segment defines the scope of all placement algebra clauses specified within it. Therefore, all icon types specified within a given *RELATIONSHIP* clause correspond to the possible icon types participating in the *sending* and *receiving* roles of the specified edge type. The *grouping* construct is a rule that results in the formation of a group of icons. The grouping construct allows members of a group of icon types to be placed relative to each other accordingly.

The placement of the icon is specified in terms of the *placement algebra* segment. The *placement_statement_list* is a list of placement algebraic statements. These placement algebraic statements include the icon types involved in the current *CASE* block within the current *RELATIONSHIP* block. The statements are conjunctive in nature. That is, the placement algebraic statements can have more than one placement constraint imposed on the participating icons.

The *icon_icon_placement* rule is used to define the explicit placement of one icon relative to another according to the placement algebra defined in Section 4.3.

The *conflict_resolution* segment provides the diagram modeler with the capability of resolving certain conflicts within the placement constraints. A typical conflict is, if icon a and b were involved in two or more relationships in which icon a was required to be left_of icon b for one of the relationships, and right_of icon b for another relationship. The conflict resolution code would allow a modeler to foresee this conflict and relax

```
RULE_BLOCK
    ASSOCIATE ent: entity_icon;
    ASSOCIATE rel: relationship_icon;

    RELATIONSHIP association_uni_flow(_,_)
        CASE ent, rel:
            BELOW(ent,rel);
        ENDCASE
    ENDRELATIONSHIP
    RELATIONSHIP is_supertype_uni_flow(_,_)
        CASE ent, rel:
            ABOVE(ent,rel);
        ENDCASE
    ENDRELATIONSHIP
    RELATIONSHIP binds_uni_flow(_,_)
        CASE rel,ent:
            RIGHT_OF(rel,ent) AND ABOVE(rel,ent);
        ENDCASE
    ENDRELATIONSHIP
ENDBLOCK
```

Figure 4.14: Icon Placement Fragment of a DLL Program for WMRT Method

one of the relationship's placement constraints thus eliminating the conflict.

4.6.6.3 Examples

Example 1:

The DLL fragment illustrated in Figure 4.14 is from the modeling of a Ward and Mellor Real-Time method diagram. This fragment incorporates specific placement algebraic binary operators to specify the relative placement of icon types.

The means by which the placement algebra is incorporated is as follows. The methods engineer determines which icon types are to be used in the current rule block via the *associate* statements. The particular edge types to be used are represented in the *relationship* blocks. The methods engineer then specifies each possible *case* of participating icon types for each edge type specified in the relationship block. In the edge types used in Figure 4.14, the number of possible cases is one for all three edge types. Refer to the definition of the Ward and Mellor Real-Time method definition in Figure 4.2. All of the corresponding Metaview relationship types have only one entity type for each participant role, therefore, there could only be one possible case of participating icon types for each edge type. The methods engineer then specifies how the participating icon types are to be placed relative to one another for the current edge type. Consider the last relationship block in the code. The edge type referred to in this block is *binds_uni_flow*. The one possible case of participating icon types is *relationship_icons* as the sender participant and *entity_icons* as the receiver participant. The placement algebra code below the *case* statement specifies that for all instances of the edge type *binds_uni_flow*, its *sender* participant, a *relationship_icon*, will always be placed to the *right_of* and *above* the *receiver* participant, an *entity_icon*.

Figure 4.15 illustrates a possible layout of the specification database of Section 4.2.2 which adheres to the DLL code fragment's placement constraints.

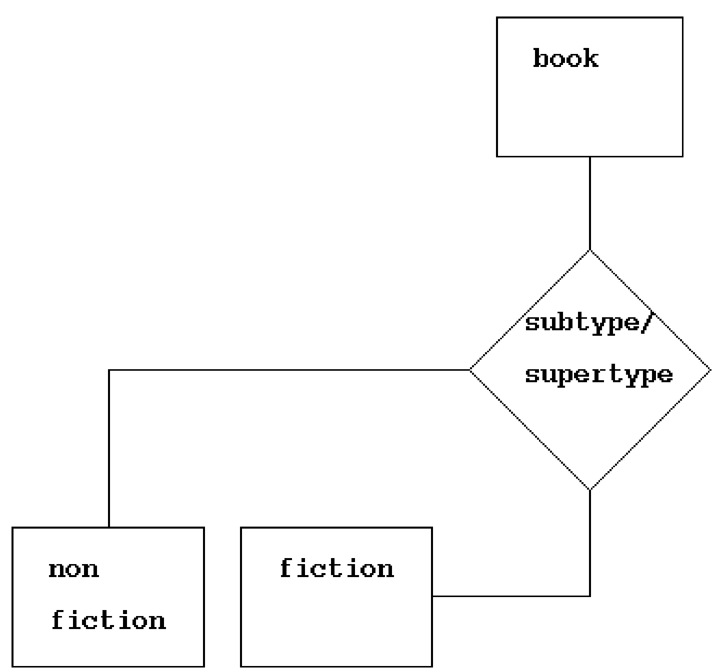


Figure 4.15: Possible ADL-generated Entity-Relationship Diagram

Example 2:

The DLL fragment in Figure 4.16 is from a Data Flow Diagram modeling program. This code illustrates how a methods engineer handles the problems of direct conflicts.

The code fragment illustrates that a possible *direct* conflict exists for the edge type *uni_flow_sends*. Direct conflicts can occur in the icon placement section of a DLL program whenever the Metaview method definition has the potential for different entity types to participate in the same role of a relationship type. For example, the relationship type that corresponds to the edge type *uni_flow_sends* has three different combinations of participating entity types. These three combinations involve two entity types which correspond to the icon types *interface_icon*, and *process_icon*. Both roles of the relationship type can have either entity type as its participant. The direct conflict in this case occurs when the *sender* and *receiver* icon types are both the *process_icon*. In this case, it is possible for instances of *process_icons* to have placement constraints relative to each other in such a way that they are in direct conflict. For example, a specification database has two instances of the relationship type which corresponded to the *uni_flow_sends* edge type. The first participant for the first instance of the relationship type was process a, and the second participant was process b. The first participant for the second instance of the relationship type was process b, and the second participant was process a. The placement algebraic rules coded in the *RELATIONSHIP* block of the code would lead to the generation of directly conflicting object placement constraint code from the ADL Tool for a given specification database. For the first instance of the relationship type, *process_icon* a would be *right_of process_icon* b and for the second instance of the relationship type, *process_icon* b would be *right_of process_icon* a. These constraints clearly can't be met simultaneously, therefore one has to be relaxed, or removed. This is where the constraint resolution code allows the methods engineer to focus on the potentially offending edge types and the possible conflict and then choose which placement constraint should be removed and which should be kept. The code handles the prob-

```
RULE_BLOCK

    ASSOCIATE int: interface_icon;
    ASSOCIATE pro: process_icon;
    ASSOCIATE ds: datastore_icon;

    RELATIONSHIP uni_flow_sends(_,_)
        CASE int,pro:
            LEFT_OF AND ABOVE;
        ENDCASE
        CASE pro,int:
            RIGHT_OF(pro,int) AND BELOW(pro,pro);
        ENDCASE
            CASE pro,pro:
            RIGHT_OF(pro,pro);
        ENDCASE
    ENDRELATIONSHIP

        ⋮       some icon placement code omitted for clarity

    CONFLICT:
    {
        uni_flow_sends(pro,pro);
    }
    RESOLUTION_MAPPING:
    {
        uni_flow_sends(SENDER) = uni_flow_sends(RECEIVER):
            PRIORITY_IS uni_flow_sends(SENDER);
    }

ENDBLOCK
```

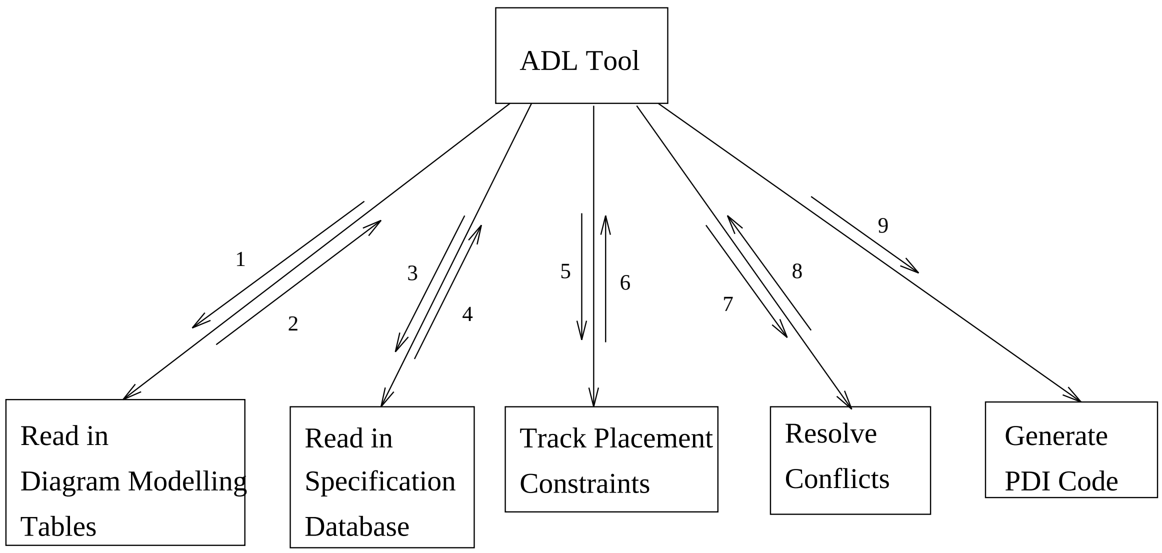
Figure 4.16: Conflict Resolution Fragment for Data Flow Method

lem by, first, focusing on the edge type which might lead to a possible conflict and, second, specifying what the condition for a conflict would have to be. In this case, when the *sender* participant of a *uni_flow_sends* edge type instance equals the *receiver* participant of an instance of the same edge type, a possible placement conflict could occur. This example solves the potential conflict by prioritizing the first participant of the *uni_flow_sends* edge type. Therefore, for the example involving *process_icon a* and *process_icon b* the conflict resolution code favors the placement of *process_icon a right_of process_icon b*, and does not allow the generation of the placement constraint *process_icon b right_of process_icon a*.

4.7 The ADL Tool

The ADL Tool is the second component of the ADL system, as shown in Figure 4.1. The ADL Tool exists at the Metaview Meta level as a generic Metaview tool. The generic ADL Tool was developed at the Metaview meta level by a Tool Definer. It is composed of the algorithms for the generation of the layout code, but doesn't have the diagram layout tables/rules. The generic ADL Tool is configured for a specific diagram type at the environment level by a methods engineer by associating it with a specific set of diagram modeling tables. The configured ADL Tool is then considered to be at the environment level. The generic ADL Tool has all the same components of a configured ADL Tool, except for the diagram modeling tables. The generic ADL Tool, thus, forms the core of a configured ADL Tool. The developer, at the user level, then provides the configured ADL Tool with the name of a specification database to be laid out. The configured ADL Tool reads in the given specification database, processes the placement rules, resolves foreseen conflicts (according to the diagram type modeling tables), and generates the appropriate PDI code to have ICOLA lay out the diagram.

The discussion thus far has focused on the generic versus the configured ADL Tool. The structure of the ADL Tool, generic or configured, is constant. This structure is illustrated in Figure 4.17. This figure illustrates the functionality of the ADL Tool in the form of a Yourdon [29] structure chart.



Legend:

1: name of diagram modeling tables file
2: diagram modeling tables
3: name of specification database, icon type and edge type tables
4: icon and edge instance tables
5: icon type placement table, icon and edge instance tables
6: icon instance table
7: icon and edge instance tables, conflicting edge types table, conflict resolution table
8: icon instance table
9: icon and edge instance tables, output file name

Figure 4.17: Structure Chart of the ADL Tool

4.8 Edge Routing

The Diagram Layout Language and the ADL Tool, handle the problem of icon placement. However, the problem of placing or *routing* the edges between the icons still remains. The Diagram Layout Language allows the methods engineer to dictate what relationship types (edge types) are to be represented on the diagram, but it does not allow him/her to dictate more than just the very basic graphic properties of edges on the diagram. The edge router presented in this section represents the third component of the ADL system as shown in Figure 4.1.

Edge routing is not a trivial problem. Raghavan [21] showed that general rectilinear (grid) routing, subject to constraints similar to those found in automatic diagram routing, is an NP-hard problem.

4.8.1 The A* Algorithm

The approach chosen for this thesis to perform edge routing is based on circuit design automation. One such approach is a search algorithm developed by artificial intelligence research. This algorithm is known as the A* algorithm (Tanimoto [26]). It is similar to the breadth-first search except for the following difference. This algorithm uses a priority queue instead of a breadth-first queue to search possible paths. The A* algorithm essentially prunes a search tree by expanding nodes which have an acceptably low estimate of the cost to get to the destination. This cost is referred to as the priority. The lowest *priority* position is the next one examined. This continues until the destination is reached. To guarantee the cheapest path, the estimated cost to a destination should always be higher than what it really is, therefore, it is overestimated.

This algorithm assumes a Cartesian coordinate grid system in which each pair of coordinates corresponds to a cell or node. This is the reason why DLL, of section 4.6.3.2, only allows the routing standard to be grid.

The A* search algorithm works by initializing the two-dimensional grid, which forms the search space. Once the grid has been initialized, the A* algorithm determines the source and destination cells for a given edge to be routed. The grid initialization and determination of which edge to route is performed with information retrieved directly from ICOLA. A* then performs its search by processing a queue of open cells, as in Lee's algorithm [16], which initially contains only the source cell. But this is a priority queue, which means cells are inserted according to an assigned priority, not just at the end as in a typical FIFO queue.

The priority is based on the cost so far of the current position plus the *estimated cost to destination*. Cells that have the lowest priority are placed at the head of the queue. Cells are still removed from the head of the open queue, then they are checked to see if they are the target cell, and if they are not, their neighboring cells are put on the open queue at the proper position. Neighboring cells which have already been searched are checked to see if the new path between them and the source cell is better than the previous one. If the new path is better, the neighboring cells are repositioned on the open queue according to the newly determined priority. As in the breadth-first search, this continues until the target cell has been found or the open queue is empty. Once the search has been concluded, the A* algorithm returns either the routing information (placement information for the routed edge) or an error message if the edge couldn't be routed.

As in Lee's algorithm, if a neighboring cell is classified as blocked, then that neighboring cell is not placed on the open queue. Blocked means that there is something in the neighboring cell through which a path can't be routed. Basically, each neighboring cell is scrutinized according to certain diagramming aesthetic criteria and a decision is made as to whether the cell should be enqueued or not. It is in this scrutinization that criteria for diagram layout are employed. The criteria are as follows:

- All routed edges should be a reasonable distance away from icons and should NOT pass through any icons.
- All routed edges should not be routed too close to other edges and should not be routed, either partially or entirely, on another routed edge's path.
- An edge may cross another edge.

4.8.2 The Heuristic

The A* algorithm depends on being able to estimate the distance between a cell and the target cell. This is one approach to determining the priority of the cell. In the case of autorouting, a simple measure of this distance is available, and this helps the A* algorithm to concentrate the search in the direction most likely to succeed. The more accurate the estimate, the faster is the search. In autorouting, shortest distance is important. In diagram layout, shortest distance is not as critical. However, shortest distance has provided reasonable results in edge routing, as will be shown in Chapter 6, thus the shortest distance heuristic was used.

In practice, The A* algorithm does not suffer from the quadratic behavior of Lee's algorithm, so it solves the same problems faster, and can be applied to the larger problems that Lee's algorithm performs so poorly on. The A* algorithm does not suffer from the complexity problems of Lee's algorithm because the chosen heuristic tends to force the search in the right direction, thus pruning a large portion of the search space. As the distance between the source and target cells increases, the number of cells processed by the A* algorithm will increase, but not as dramatically as with Lee's algorithm.

4.9 ADL System Design Summary

The ADL system is composed of three main components:

- DLL Compiler — This component is responsible for accepting a DLL program as input and generating diagram modeling tables.
- ADL Tool — This component, once configured for the appropriate method, is responsible for reading in a Metaview specification database, the diagram modeling tables and the appropriate icon definitions and then generating the appropriate ICOLA PDI code, which, when input to ICOLA, will result in the placement of the diagram's icons. The diagram's icons correspond to instances of entity types which reside in the Metaview specification database.
- Edge Router — This component is responsible for determining appropriate paths for the edges which correspond to instances of relationship types which reside in the Metaview specification database.

The ADL system is used by both methods engineers and developers. The methods engineers code a DLL program to model a diagram type which corresponds to a Metaview method. The methods engineer then configures the ADL Tool for use with a specific Metaview method. The configured ADL Tool can then be used by developers to generate the ICOLA PDI layout code for many different specification databases.

These PDI code files are then fed into ICOLA, the current object platform, to produce the actual diagram.

CHAPTER 5

Implementation of the ADL System Prototype

5.1 Introduction

The implementation of the prototype of the ADL system is discussed in detail in this chapter. In Chapter 4, the ADL system was discussed at the design level. The DLL compiler and ADL Tool were responsible for the basic placement of the icons whereas the routing of edges was handled by the A* routing algorithm.

The ADL system prototype currently consists of a DLL compiler, an ADL Tool, an icon library, and an edge router incorporated within the ICOLA platform. The DLL compiler, ADL Tool, and the A* edge router each consist of about 2000 lines of standard C code. The icon library currently consists of ten icon types which are encoded in ICOLA's PDI language.

The rest of the chapter is organized as follows. First, Section 5.2 describes the DLL compiler. The ADL Tool is presented in Section 5.3. The chapter concludes with a description of the A* edge router.

5.2 The DLL Compiler

Figure 5.1 illustrates a breakdown of the compiler into four basic modules. These modules are the scanner, the parser, the symbol table handler and the tables generator. The scanner reads the DLL program as input, and returns a token representation of the input to the parsing module. The parser performs the syntactic analysis of the DLL source program. The routines of the symbol table handler look after insertion, retrieval and deletion of identifiers from the symbol table. The tables generator generates the diagram modeling tables which will be used by the ADL Tool. The parser and the tables generator both make use of the symbol table.

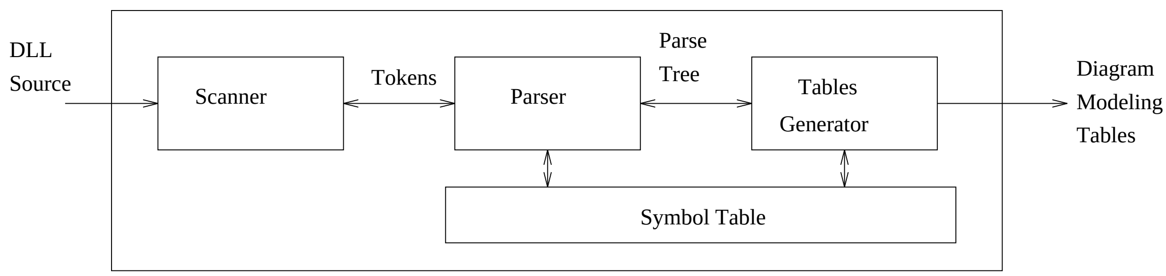


Figure 5.1: Organization of the DLL Compiler

5.2.1 Overview of the Attributed Translation System (ATS)

The DLL compiler implementation requires the ATS (Attributed Translation System) parser generator[3]. ATS is a system for generating attributed parsers for LL(1) attributed grammars. The ATS system, is a compiler writing tool which translates a left-attributed translation grammar (LATG) specification for a language into an LATG parser for that language. The compiler writer using ATS supplies the attributed LL(1) input grammar, scanner, and all necessary action routines in order for ATS to generate the appropriate parse tables and parser. The organization of an ATS-generated compiler is illustrated in Figure 5.2. In the case of DLL, ATS translates the input grammar of Appendix A into a sequence of actions, according to the user-supplied action routines, which, in turn, translate a DLL program into diagram modeling tables.

Figure 5.2 illustrates that the parser invokes the user-supplied scanner, as well as the action routines supplied by the compiler writer. The error recovery routines have defaults provided by ATS, but those too can be tailored by the compiler writer. The details of generating a compiler via ATS are dealt with in detail in Bocking [3].

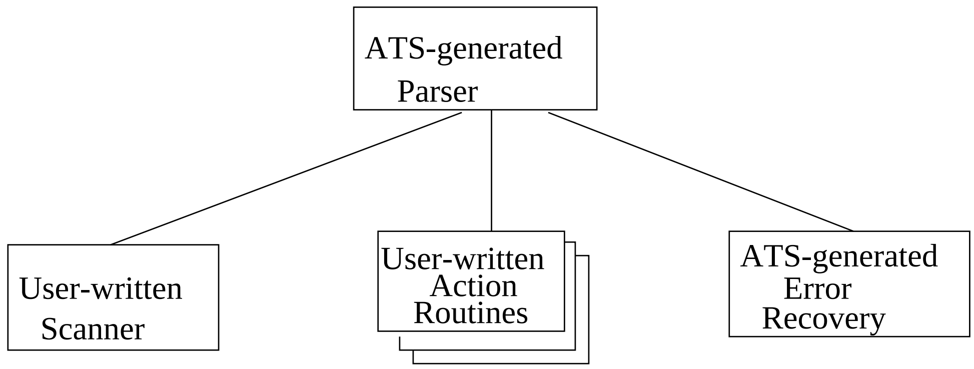


Figure 5.2: Organization of a Typical ATS-generated Compiler

The DLL compiler of Figure 5.1 and the ATS generated compiler organization chart of Figure 5.2 represent essentially the same thing. Figure 5.1 views the DLL compiler from the functional point of view, whereas Figure 5.2 views the DLL compiler from the point of view as to how it's implemented in ATS. In Figure 5.1, the symbol table handler and tables generator correspond to Figure 5.2's *user-written action routines* whereas the scanner in Figure 5.1 is equivalent to the scanner in Figure 5.2. The ATS-generated error recovery of Figure 5.2 is built into ATS and is based on the error recovery technique of Fisher, Milton and Quiring [7].

5.2.2 The Scanner

It is the DLL scanner's responsibility to recognize and categorize lexical units or tokens in a DLL program. Tokens could be any of the following, *identifiers*, *symbols*, *numbers*, and *keywords*. The scanner is invoked by the parser each time the parser needs a new look-ahead token. A scanner recognizes complete tokens, which are sent to the parser, and ignores other tokens such as blanks, tabs, new line characters, and comments. The scanner generates appropriate error messages for invalid tokens.

The scanner is essentially a finite state machine, and its operation can be most easily explained by representing the scanner by a finite state diagram. Figure 5.3 is a finite state representation of the lexical scanner for DLL.

The DLL scanner reads characters from a DLL source file. If the character represents a single-character token then the corresponding token is sent to the parser. The action of sending a token to the parser is represented by an acceptor state (double circle icon) in Figure 5.3. If the character is an invalid character or part of a comment then the scanner ignores it and continues reading in characters from the source file until a valid character or end of file (eof) character is encountered. If the character is a digit, then more characters are read until a non-digit character is read. The *NUMBER* token is sent to the parser. If the character is a letter, then more characters are read until a non-valid character is read. The scanned word is then compared to a list of keywords. If it is a keyword, then the appropriate *KEYWORD* token is sent to the parser, otherwise an *IDENTIFIER* token is sent to the parser.

Whenever the scanner recognizes one of the tokens defined in the finite state diagram, it returns the token to the parser. The parser then uses this token to determine the correctness of the syntax of the source DLL program.

5.2.3 The Parser

The purpose of the parser is to determine whether or not a given DLL program follows the syntax rules for the Diagram Layout Language described in Chapter 4.

ATS is a "top-down" parser generator system. To accommodate a top-down parse, the grammar representing the syntactic rules of DLL must be in LL(1) form. LL(1) grammars parse in O(n) time; where n represents the number of input tokens. A more complete treatment of attributed translation and LL(1) grammars can be found in Tremblay and Sorenson [27, 28]. For presentation and ease of understanding purposes, the DLL description grammar presented in Chapter 4 was not necessarily in LL(1) form. The LL(1) version of this presentation grammar, which is presented in Appendix A, also has associated action routines. Some of these routines are used for symbol table management and tables generation. The following sections describe these routines.

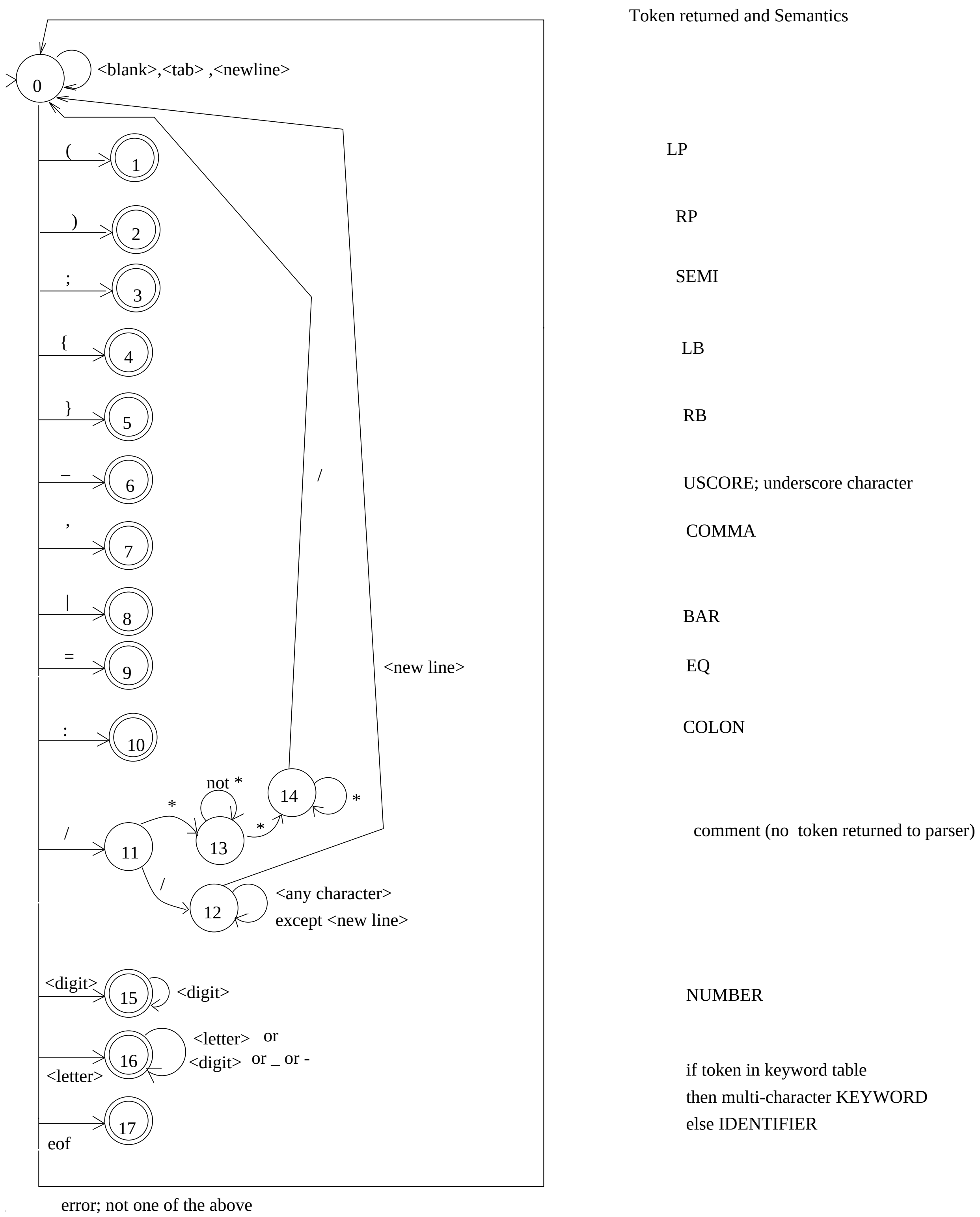


Figure 5.3: Finite State Machine Representation of the DLL Scanner

Table 5.1: Symbol Table Record Structure

identifier name	Icon Name	hashing link

5.2.4 The Symbol Table

A stack-implemented hash-structured symbol table was implemented. This type of symbol table is presented and discussed in Tremblay and Sorenson [27] on pp. 471-474. There are three main data structures in this type of symbol table. These components are the stack, the block index and the hash table. Whenever a rule block is entered, the local identifiers in that rule block are pushed onto the stack and the location of the rule block's first identifier on the stack is placed in the block index. Whenever a rule block is exited, the previous steps taken upon entering a block are undone. The rule block's identifiers are popped off the stack and the rule block's index number is removed from the block index. The hash table is used to quickly access the identifiers stored on the stack. Like the stack and the block index, the hash table must be updated whenever a block is entered or exited. Details on this update are given in the discussion of the symbol table operations.

Since DLL is a block-based language, the block index is an essential component in managing the symbol table.

5.2.4.1 Symbol Table Data Structures

The hash table, block index, and the symbol table stack itself are all implemented as arrays. The hash table is implemented as an array of integers named *hash[i]*. Its maximum size is determined by the constant *HSIZE* which is currently set to 101. The block index is implemented as an array of integers named *block[i]*. Its maximum size is determined by the constant *BSIZE*, which is currently set to 5. The symbol table stack is implemented as an array of record and is called *stack[i]*. The maximum size of the stack is determined by the constant *SSIZE*, which is currently set to 200.

DLL requires certain information to be stored about each identifier in the symbol table. Table 5.1 outlines the information which is kept for each identifier.

Each record help on the stack consists of three fields. The first is *identifier name*, a string of 256 characters that holds the identifier name. The next field holds the Metaview entity type name. The identifiers which are kept in this table are only those that are required at a future point in the compilation process, not every identifier used in a DLL program is required to be inserted into the symbol table. Essentially, only one type of identifier is placed in the symbol table. This identifier type is for the representation of icon types which are defined in the *ASSOCIATE* construct of Section 4.6.6.

The *hashing link* value is used by the symbol table management routines for resolving possible collisions while inserting and retrieving entries from the table. The collision resolution technique which is used in this implementation is known as *chaining*, and it is explained as follows. The hashing *link* is an integer which acts as a *pointer* to an element in the stack array by indicating the index of the array member to which the current array member is linked. For example, the statement *stack[4].link = 2* indicates that *stack[4]* is linked to *stack[2]*.

Because the hash table is used to reference the symbol table stack, the hash array must point to elements of the stack array in some way. Some arrays are static, each cell of the hash array holds an integer value that can be regarded as a pointer. If the value of *hash[i]* is positive, it represents the index of a cell in the stack array and so it points to the contents of that cell. If *hash[i] = 5*, then *hash[i]* points to *stack[5]*. Similarly, each cell of the block index array holds an integer value that represents the index of a cell in the stack array. In both the hash and block index arrays, the equivalent of a nil pointer is represented by the value of -1.

More pointers are used in the stack array. These pointers are the hashing *link* field which may point to another cell in the stack array. Links are necessary within the stack array in order to resolve any hash_table collisions. Hash_table collisions occur when two or more identifiers hash to the same location in the hash table. These collisions are resolved here by *chaining* identifiers that have the same hash value. The hash array cell points to the topmost of these identifiers in the stack array, while the link field of this topmost identifier gives the stack index of the next identifier

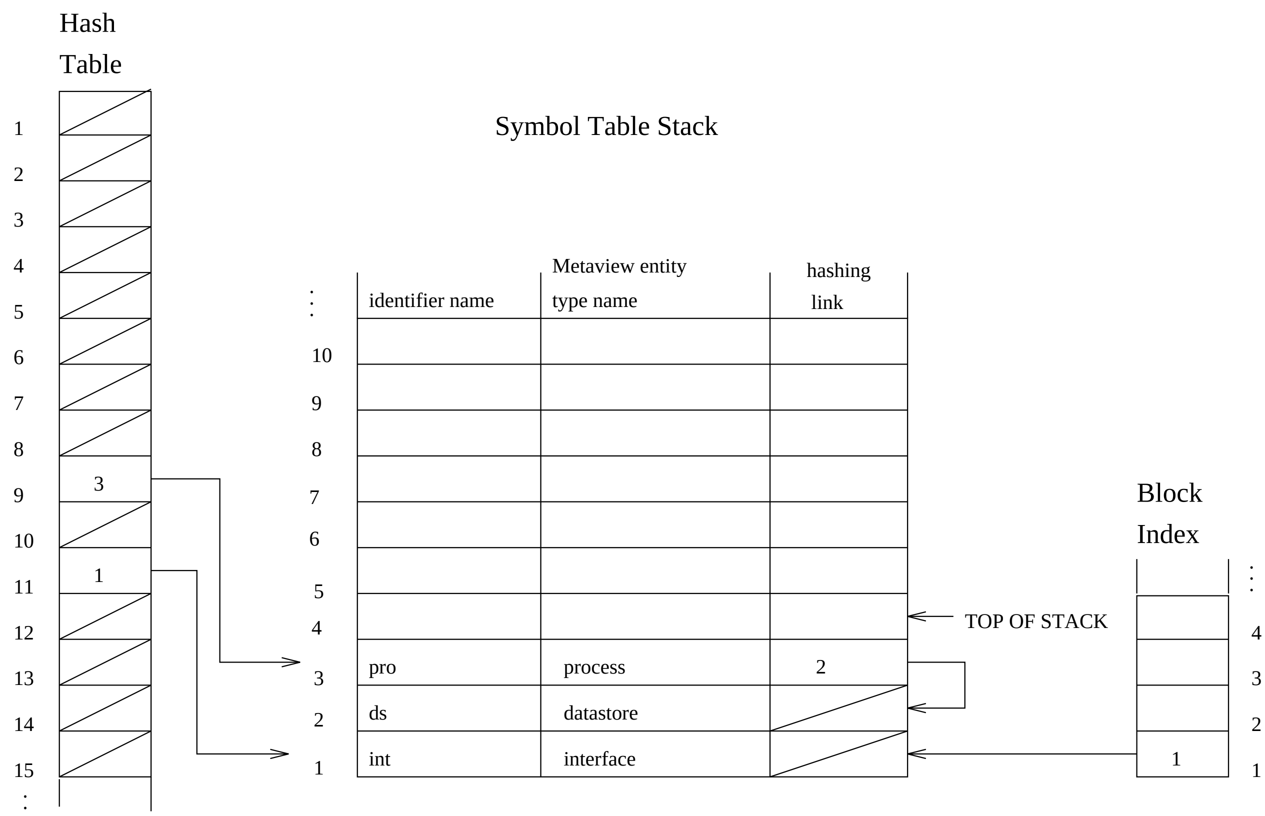


Figure 5.4: Organization of the Symbol Table's Data Structures

involved in the collision. This linking continues down the stack array until all of the *colliding* identifiers are accounted for.

For example, assume that three identifiers, *int*, *ds*, and *pro* are declared, in that order, at the beginning of a rule block within a DLL program. Assume also that *pro* and *ds* both hash to location 9 in the hash table, whereas *int* hashes to location 11. This would require the stack record for *pro* to have the *hashing link* field to point to the stack record containing the *ds* identifier. Figure 5.4 illustrates this example.

5.2.4.2 Symbol Table Management Action Routines

There are several operations used in the management of DLL's symbol table. The symbol table is managed by action routines incorporated within the grammar. Action routines can be thought of as pseudo-non-terminal symbols of the grammar. They are not expanded like real non-terminal symbols of the grammar, however, they are invoked as the parser reaches that portion of the right-hand side of a production as a left-hand side non-terminal is being expanded. Action routines are subroutines which are incorporated into the DLL grammar. Therefore, one can visualize the action routines within the grammar similarly to the other symbols of the grammar within a parse tree for a given production. The DLL compiler's parser invokes these action routines as the parser processes the DLL source. A complete listing of the DLL input grammar, with accompanying action routines, is given in Appendix A. Because action routines are used, a notation to explain the interface of the action routines to the DLL parser follows.

Notation

This section outlines the notation used to explain the action routines used and how they interface to the DLL compiler.

We denote an action routine within the ATS input grammar as follows:

action_routine_name < list_of_inherited_attributes > list_of_synthesized_attributes

The inherited attributes are the inputs to the action routine and the synthesized attributes are the output of the action routine. Consider the following ATS production rule which incorporates an action routine:

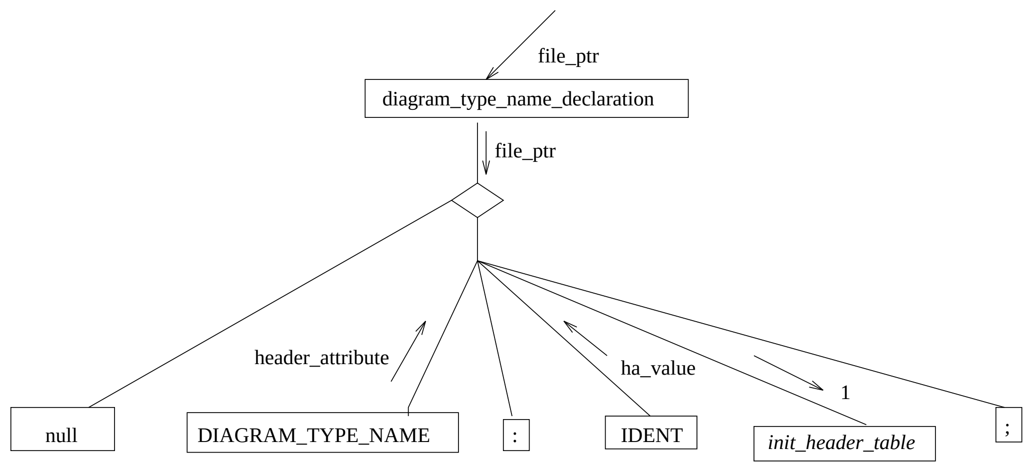
diagram_name_declaration<file_ptr:

ϵ | DIAGRAM_NAME>header_attribute COLON IDENT>ha_value

init_header_table<header_attribute,ha_value,file_ptr SEMI;

The action routine, *init_header_table*, has the inherited attributes *header_attribute*, *ha_value*, and *file_ptr*, and no synthesized attributes. The terminal symbols DIAGRAM_NAME and IDENT both synthesize one attribute which are inherited into *init_header_table* along with the attribute *file_ptr* which is inherited from higher nodes in the parse tree. The symbol ϵ denotes the empty string or empty production.

Figure 5.5 illustrates the above production rule. Inherited attributes are represented by a downward directed arrow with accompanying attribute names. Synthesized attributes are represented by an upward directed arrow with accompanying attribute names. Non-terminal symbols, terminal symbols, and action routines are



Legend:

1: header_attribute, ha_value, file_ptr

N.B. Grammar non-terminal symbols are represented by lowercase words, terminal symbols are represented by uppercase words, and action routines by italicized words.

N.B. The line ·········· on this type of diagrams means that the parse tree for the non-terminal symbol being expanded is already on the diagram or on another diagram in this section and repeating it will cause the diagram to lose clarity.

Figure 5.5: Notation Convention for DLL Compiler Action Routines

boxed. Non-terminal symbols of the grammar are represented by lowercase words. Terminal symbols are represented by punctuation symbols or uppercase words. Action routines are represented by lowercase, italicized words. The diamond icon represents alternate right-hand sides of the production.

For example, the action routine *init_header_table* is explained as follows:

init_header_table<header_attribute,ha_value,file_ptr

Function: This routine appends one record to the general diagram header table using the *header_attribute*, and *ha_value* input attributes to fill in the appropriate fields in the header table record. The record is written to the file pointed to by the attribute *file_ptr*. Figure 5.5 illustrates how the action routine *init_header_table* interfaces to the DLL parser.

Symbol Table Management Action Routines

The symbol table action routines are presented as follows.

Init

Function: This parameterless operation initializes the hash table, stack, and block index to empty. This operation is invoked at the beginning of the parsing process of a DLL program.

Set

Function: This parameterless operation is invoked at the beginning of a new block. It simply sets the next block index to point to the current top of the symbol stack. This operation is invoked as an action routine at the beginning of a *RULE_BLOCK* clause.

Reset

Function: This parameterless routine is invoked at the end of a block. The routine pops the identifiers belonging to the top block off the symbol table stack. It also updates all hash table pointers and removes the top block from the block array. This operation is invoked at the end of a *RULE_BLOCK* clause.

Duplicate(identifier)

Function: This routine, which is invoked by the *insert* operation, returns *true* if the provided *identifier* is already in the symbol table of the current block. It is not an action routine but it is mentioned here for completeness. Its input parameter is the name of the identifier. The output parameter is a boolean value, *true* or *false* which specifies whether or not the identifier is in the symbol table.

Insert<identifier

Function: This routine calls *Duplicate* to determine whether the *identifier* already exists within the currently active block, i.e., the top block. If no duplicate is found in the currently active block, the name of the identifier is inserted into the table. Insert is called in the *ASSOCIATE* clause of the grammar. The interface to the grammar is illustrated in Figure 5.6.

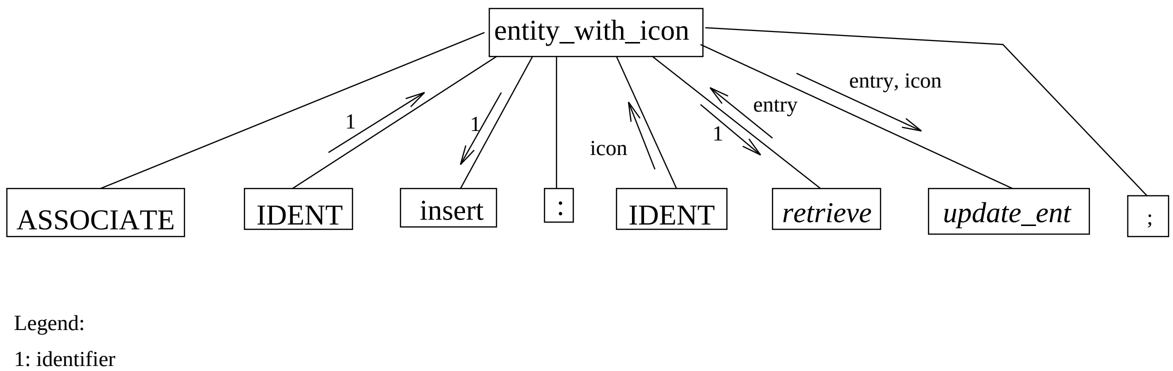


Figure 5.6: The ASSOCIATION Clause Structure

Retrieve<identifier>entry

Function: This routine returns the index, attribute *entry*, into the symbol table stack of the first occurence of the given *identifier*. The search for the identifier is performed by first accessing the hash table and then, if necessary, by traversing the chain formed by the link fields of colliding identifiers. This action routine is invoked at several points in the grammar. These points are in the *ASSOCIATE* clause illustrated in Figure 5.6, the *CASE* clause of Figure 5.18, and in the *CONFLICT* clause shown in Figures 5.19 and 5.20.

Update_ent < entry, icon

Function: This routine is responsible for updating the *Metaview entity type name* field with the value of the *icon* input attribute, at record number *entry* of the symbol table stack. This operation is invoked as an action routine in the *ASSOCIATION* clause of the grammar as illustrated in Figure 5.6.

Hash

Function: The hashing function used is:

$$H(x) = ((x[0] + x[last]) + 16 \times sizeof(x)) \text{ modulo (size of hash table)}$$

where $x[0]$ is defined as the ascii value of the first character in string x and $x[last]$ is defined as the ascii value of the last character in string x. With this hashing function, any strings with the same first and last characters and same size will hash to the same

Table 5.2: General Diagram Header Table Structure

header attribute	header attribute value	header attribute value string

position in the hash table. This hashing function, determined by trial and error, gave reasonably uniform results. It is not an action routine but it is mentioned here for completeness.

5.2.5 Tables Generation

The DLL compiler translates a source program into diagram modeling tables. These tables represent the diagram layout rules encoded in the statements of a DLL program. The tables are the output of the DLL compiler. The DLL compiler appends all tables onto the same file for the sake of simplicity. Each record appended to the file has a table identification number prefix which lets the ADL Tool know which table the current record belongs to.

5.2.5.1 The Data Structures

There are six diagram modeling tables that the DLL compiler generates. Not all six tables are necessarily generated. The details of which tables will be generated for a given program is discussed in Section 5.2.5.2. The six tables generated correspond to the four main sections of a DLL program: the general header section, the graphical definitions section, the Metaview connection section, and the icon placement section. The six tables are structured as follows.

1. General Header Table — This table records the information encoded in the general header section of a DLL program. This information is: diagram type name, routing standard, and diagram type orientation. The record structure for this table is illustrated in Table 5.2. The *header attribute* records the token of a diagram header attribute, for example, *ORIENTATION*. The other header attributes are *DIAGRAM_TYPE_NAME* and *STANDARD*. The token referred to is a number which

Table 5.3: Sample General Diagram Header Table

DIAGRAM_TYPE_NAME		wmrt
ORIENTATION	TOP_DOWN	
STANDARD	GRID	

Table 5.4: Icon Table Structure

icon type	label status	entity type	label source

uniquely identifies the header attribute. For each header attribute, there is a *header attribute value* and a *header attribute value string*. Header attributes can have their values recorded as tokens(numbers) or as a string in the *header attribute value* and *header attribute value string* fields, respectively. For example, for a DLL program with the following general diagram header,

PROGRAM_NAME: test;

DIAGRAM_TYPE_NAME: wmrt;

ORIENTATION: TOP_DOWN;

STANDARD: GRID;

the DLL compiler would generate a header table as shown in Table 5.3. All the capitalized words correspond to the tokenized keywords of DLL and the lowercase words are literal strings.

2. Icon Table — This table records the information regarding the graphic properties of icon types encoded in a graphic definitions section. The table also records information about the mapping between the icon types and their corresponding Metaview entity types encoded in the Metaview connection section. The record structure for this table is illustrated in Table 5.4. The *icon type* field holds the name of the DLL-encoded icon type name. The *label status* field records a numeric token which is either *LABELED* or *NON-LABELED*. The *entity type* field records the name of the Metaview entity type which is mapped to the icon type. The *label source* field records the name of the Metaview entity type's attribute whose value should be used as the label of the

Table 5.5: Sample Icon Table

entity_icon	LABELED	object_type	default
relationship_icon	LABELED	relationship	relationship_name

Table 5.6: Edge Table Structure

edge type	label status	direction status	relationship type	label source	participant one	participant two

placed icon assuming that the icon is labelled. If the default label is desired then the word *default* appears in the label source field. As an example, for the following graphics definition and Metaview connection sections,

```
ICON_TYPES: {LABELED (entity_icon, relationship_icon)};

        ⋮            some code omitted for clarity

MAP_ENTITY entity_icon TO object_type;
MAP_ENTITY
    relationship_icon TO relationship;
    LABEL IS { relationship_name }
```

the DLL compiler would generate an icon table as shown in Table 5.5.

3. Edge Table — This table records the information regarding the graphic properties of edge types encoded in a graphic definitions section. The table also records information about the mapping between the edge types and their corresponding relationship types encoded in a Metaview connection section. The record structure for this table is illustrated in Table 5.6. The *edge type* field holds the name of the DLL edge type identifier name. The *label status* field records a numeric token which is either *LABELED* or *NON-LABELED*. The *direction status* field records a numeric token which is either *NON_DIRECTED*, *UNI_DIRECTIONAL*, or *BI_DIRECTIONAL*. *NON_DIRECTED* means that the edge type has no arrowhead(s), *UNI_DIRECTIONAL* means that there is an arrowhead on the end of the routed edge which is connected to the re-

```
ARC_TYPES: {NON_LABELED NON_DIRECTED (association_uni_flow,
                 is_supertype_uni_flow, binds_uni_flow) };

        ⋮              some code omitted for clarity

MAP_REL
    association_uni_flow TO association;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    is_supertype_uni_flow TO is_supertype;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
MAP_REL
    binds_uni_flow TO binds;
    SENDER IS ROLE 1 RECEIVER IS ROLE 2
```

Figure 5.7: Sample Program To Illustrate Edge Table

ceiver icon. *BI_DIRECTIONAL* means that there is an arrowhead on both ends of the routed edge. The *relationship* type field records the name of the Metaview relationship type which is mapped to the edge type. The *label source* field records the name of the Metaview relationship type's attribute name whose value should be used as the label of the placed icon assuming that the icon is labelled. If the default label is desired then the word *default* appears in the label source field. The *participant one* and *participant two* fields refer to the positions in the participant list of a Metaview relationship type whose entities fulfill the roles of *sender* and *receiver*, respectively.

Some Metaview relationship types have more than two roles to be filled by entity types. Considering that the ADL system can only handle binary relationships, it is important for a methods engineer to be able to pick which two roles will fulfill the sender and receiver roles for the purpose of edge routing. This concept is explained in Section 4.6.5. For example, consider the program with the graphics definition and Metaview connection sections of Figure 5.7. The DLL compiler generates the edge table shown in Table 5.7.

4. Icon Placement Table — This table records the information regarding the relative

```
RULE_BLOCK
    ASSOCIATE ent: entity_icon;
    ASSOCIATE rel: relationship_icon;

    RELATIONSHIP association_uni_flow(_,_)
            CASE ent, rel:
                    BELOW(ent,rel) AND RIGHT_OF(ent,rel);
            ENDCASE
    ENDRELATIONSHIP
ENDBLOCK
```

Figure 5.8: Sample Program To Illustrate Icon Placement Table

placement of icon types, the placement algebra, encoded in the icon placement section. The record structure for this table is illustrated in Table 5.8. The *edge type* field records the name of the DLL-encoded edge type. The *participant one* and *participant two icon type* fields record the name of the participating icon types of the sender and receiver roles, respectively, in the given edge type. The *binary placement operator* field is a numeric token corresponding to one of the tokenized binary placement operators of Table 4.3. Each record in the icon placement table corresponds to one conjunct of a *CASE* statement within the icon placement section. For example, consider the program with the icon placement section in Figure 5.8. The DLL compiler generates the icon placement table shown in Table 5.9.

5. Conflicting Edges Tables — This table records a list of edge types with their corresponding sender and receiver icon type participants. This list of edge types represents the methods engineer's list of edge types whose participants could have possibly conflicting placement constraints generated for them. The methods engineer encodes this list within conflict resolution statements of a placement of icons section.

Table 5.7: Sample Edge Table

association_uni_flow	NON_LABELED	NON_DIRECTED	association	default	1	2
is_supertype_uni_flow	NON_LABELED	NON_DIRECTED	is_supertype	default	1	2
binds_uni_flow	NON_LABELED	NON_DIRECTED	binds	default	1	2

Table 5.8: Icon Placement Table Structure

edge type	participant one icon type	binary placement operator	participant two icon type

Table 5.9: Sample Icon Placement Table

association_uni_flow	entity_icon	BELOW	relationship_icon
association_uni_flow	entity_icon	RIGHT_OF	relationship_icon

The record structure for this table is illustrated in Table 5.10. The *edge type* field records the name of the DLL edge type identifier. The *participant one* and *participant two icon type* fields record the name of the sender and receiver participating icon types for the given edge type. Each record in the conflicting edges table corresponds to one combination of edge type and participating icon types for the *CONFLICT* statement within an icon placement section. For example, for the program with the icon placement section of Figure 5.9, the DLL compiler generates a conflicting edges table in Table 5.11.

The compiler requires the use of an internal array for the purpose of generating the conflicting edges table. This internal array is called *et* which stands for *entity table*. The *et* array is used by the compiler to keep track of the combinations of the sender and receiver participating icon types for each edge type listed in the *CONFLICT* clauses of a program. The *et* table is a two-dimensional array of icon type names that are participants for a given edge type. The row determines which role, e.g. sender (row one) or receiver (row two). The column specifies the participating entity types for the given role. For example, for the previous DLL program icon placement section, the *et* array for the *uni_flow_sends* edge type is illustrated in Table 5.12. The generation and use of this table will be explained in detail in Section 5.2.5.2.

6. Conflict Resolution Table — This table records a list of conditions for conflicts, within the edge types listed in the conflicting edges table, to occur and the corre-

Table 5.10: Conflicting Edges Table Structure

edge type	participant one icon type	participant two icon type

```
RULE_BLOCK

    ASSOCIATE int: interface_icon;
    ASSOCIATE pro: process_icon;
    ASSOCIATE ds: datastore_icon;

            ⋮ some icon placement code omitted for clarity

    CONFLICT:
    {
        uni_flow_sends( pro | int ,pro );
    }
    RESOLUTION_MAPPING:
    {
        uni_flow_sends(SENDER) = uni_flow_sends(RECEIVER):
        PRIORITY_IS uni_flow_sends(RECEIVER);
    }

ENDBLOCK
```

Figure 5.9: Sample Program To Illustrate Conflicting Edges Table

Table 5.11: Sample Conflicting Edges Table

uni_flow_sends	process_icon	process_icon
uni_flow_sends	interface_icon	process_icon

sponding methods engineer encoded solution to each one of the possible conflicts. The methods engineer encodes this list within conflict resolution statements of the placement of icons section. The record structure for this table is illustrated in Table 5.13. The *edge type 1* and *edge type 2* fields record the name of the edge types from the conflicting edge table in which a solution is provided. The *role for edge type 1* and *role for edge type 2* fields record the token of the role for each edge type. These tokens can either be *SENDER* or *RECEIVER*. When the entities fulfilling those roles

Table 5.12: Sample *et* Array

process_icon	interface_icon
process_icon	

Table 5.13: Conflict Resolution Table Structure

edge type 1	role for edge type 1	edge type 2	role for edge type 2	priority edge type	priority role

```
⋮ some icon placement code omitted for clarity

CONFLICT:
{
    uni_flow_sends( pro | int ,pro );
}
RESOLUTION_MAPPING:
{
    uni_flow_sends(SENDER) = uni_flow_sends(RECEIVER):
    PRIORITY_IS uni_flow_sends(RECEIVER);
}
```

Figure 5.10: Sample Program To Illustrate Conflict Resolution Table

for the two edge types are equal, then this means that a placement conflict could occur. Therefore, the solution is to provide the *priority edge type* and a *priority role* fields so that the placement constraint code generator (the ADL Tool) can apply this rule and relax the non-priority placement constraints. For example, for the program with the conflict resolution code in Figure 5.10, the compiler generates the conflicting edges table given in Table 5.14.

This table reflects the fact that when there are two relationships corresponding to the edge type *uni_flow_sends*, then a conflict could arise if the sender entity of one of the relationships is equal to the receiver entity of the other relationship. The table provides a solution by specifying that the receiver entity of the first relationship of edge type *uni_flow_sends* has priority. Therefore, remove the sender entity from the receiver entity's placement arrays.

Table 5.14: Sample Conflict Resolution Table

uni_flow_sends	SENDER	uni_flow_sends	RECEIVER	uni_flow_sends	RECEIVER

5.2.5.2 The Table Generation Action Routines

This section describes the *action routines* used by ATS to parse a DLL program and to generate the diagram modeling tables just described. The diagrams illustrating how the action routines interface to the parser are of the same form as that used to describe the symbol table management routines given earlier.

5.2.5.2.1 Action Routines in the Diagram Header The diagram header section of the grammar has one action routine, *init_header_table*, which is responsible for generating the diagram header table. The action routine is used in three clauses in the grammar: *DIAGRAM_TYPE_NAME*, *ORIENTATION*, and *STANDARD*. Figures 5.11, 5.12, and 5.13 illustrate how the action routine interfaces to the parser in those three clauses, respectively.

init_header_table<header_attribute,ha_value,file_ptr

Function: This routine appends one record to the general diagram header table using the *header_attribute*, and *ha_value* input attributes to fill in the appropriate fields in the header table record. The record is written to the file pointed to by the attribute *file_ptr*.

5.2.5.2.2 Action Routines in the Graphic Definitions Section The graphic definitions section of the grammar has two actions routines, *init_icon_table*, and *init_edge_table*, which are responsible for initializing the icon and edge tables, respectively. The *init_icon_table* routine is used in the *ICON_TYPES* clause of the grammar. The *init_edge_table* routine is used in the *ARC_TYPE* clause of the grammar. Figures 5.14 and 5.15 illustrate how the *init_icon_table* and *init_edge_table* action routines interface to the DLL parser in the *ICON_TYPES* and *ARC_TYPES* clauses, respectively.

init_icon_table<icon_type,label_status

Function: This routine appends one record to the icon table using the *icon_type*, and *label_status* input attributes to fill in the *icon_type* and *label_status* fields, respec-

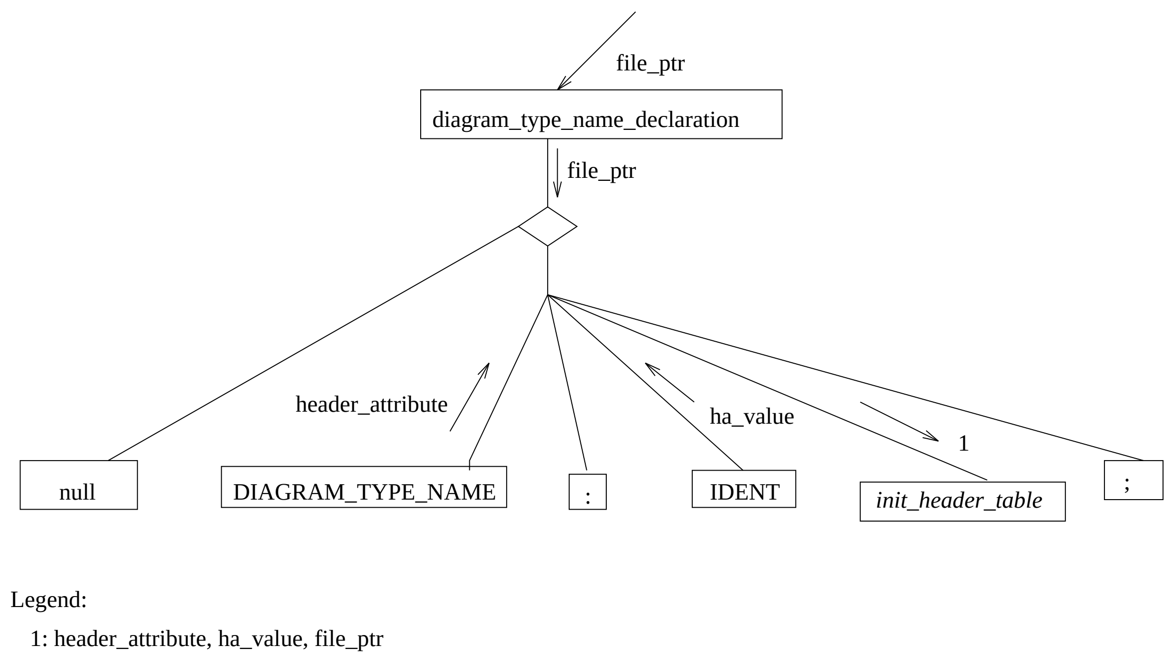


Figure 5.11: Diagram Name Declaration Structure

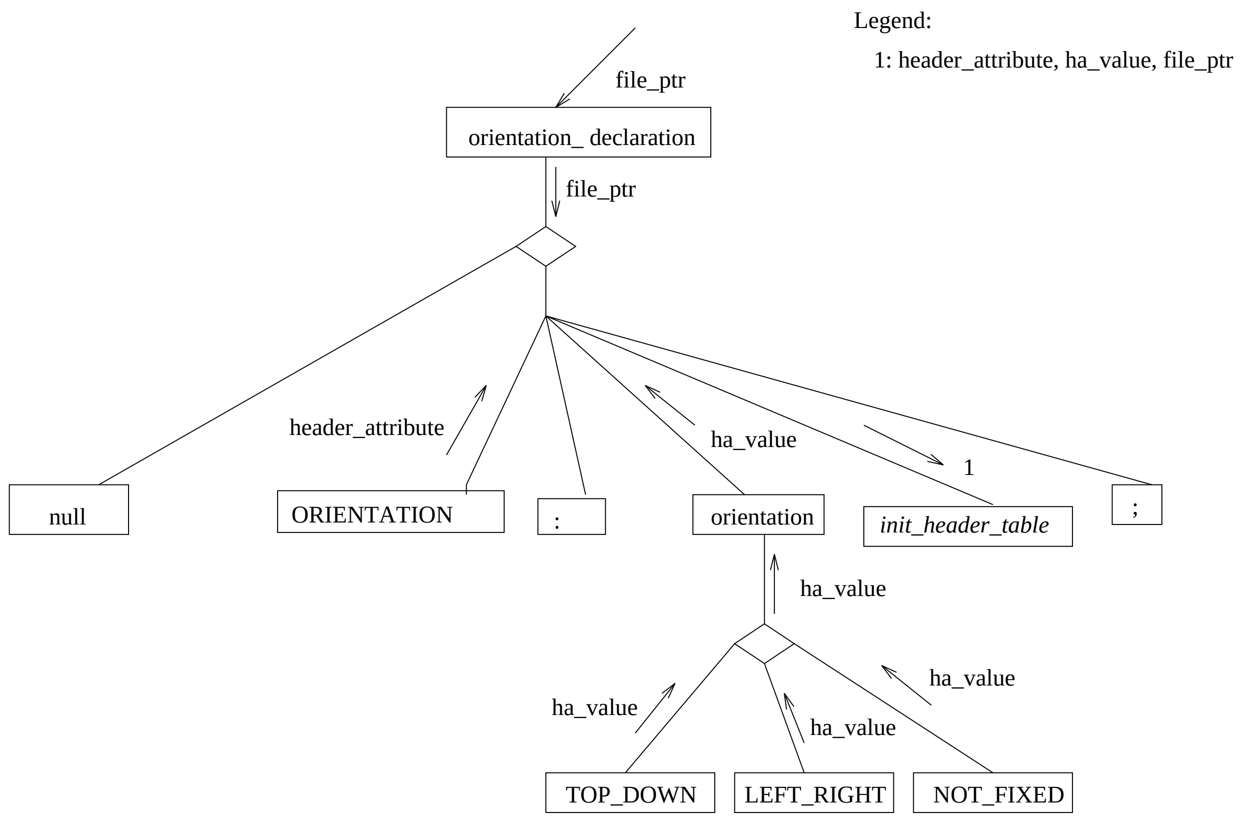


Figure 5.12: Orientation Declaration Structure

tively, in the icon table record.

init_edge_table<icon_type,label_status

Function: This routine appends one record to the edge table using the *edge*, *label_status*, and *dir_status* input attributes to fill in the *edge_type*, *label_status*, and *direction_status* fields, respectively, in the edge table record.

5.2.5.2.3 Action Routines in the Metaview Connection Section The Metaview connection section of the grammar has two main action routines, *update_icon_table* and *update_edge_table* which are responsible for updating the icon and edge tables, respectively. These tables were initialized and partially data-filled by action routines in the graphic definitions section. The *update_icon_table* is used in the *ICON_TYPES* clause (Figure 5.16) of the grammar. The *update_edge_table* is used in the *ARC_TYPES* clause (Figure 5.17) of the grammar.

update_icon_table<icon,entity,label,file_ptr

Function: This routine fills in the remaining fields of the record in the icon table whose *icon_type* field matches the *icon* attribute. The *entity_type* and *label_source* fields, of the icon table's record, are filled with the *entity*, and *label* attributes, respectively. This record is then appended to the file pointed at by the *file_ptr* attribute.

update_edge_table<edge,relationship,ep1,ep2,label,file_ptr

Function: This routine fills in the remaining fields of the record in the edge table whose *edge_type* field matches the *edge* attribute. The *relationship_type*, *label_source*, *participant one*, and *participant two* fields, of the edge table's record are filled with the *relationship*, *label*, *ep1*, and *ep2* attributes, respectively. This record is then appended to the file pointed at by the *file_ptr* attribute.

5.2.5.2.4 Action Routines in the Icon Placement Section The icon placement section of the DLL grammar has five main action routines which are responsible for generating three tables; the icon placement table, conflicting edges tables,

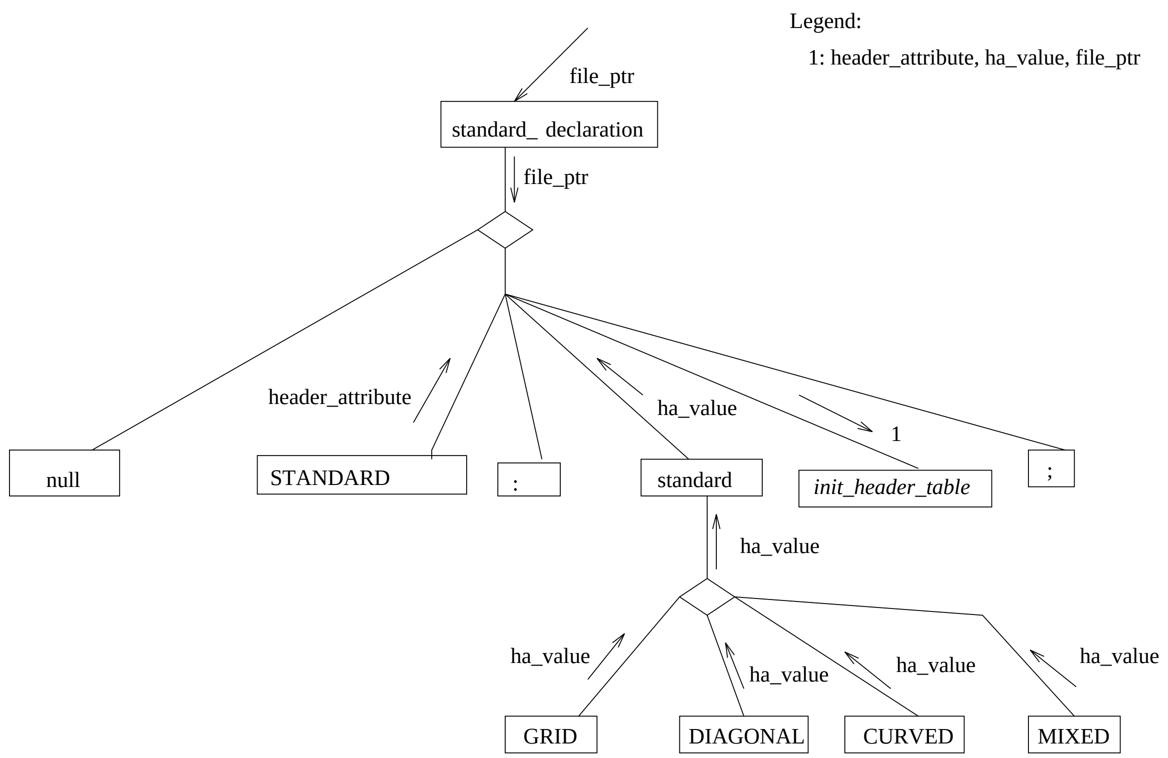


Figure 5.13: Standard Declaration Structure

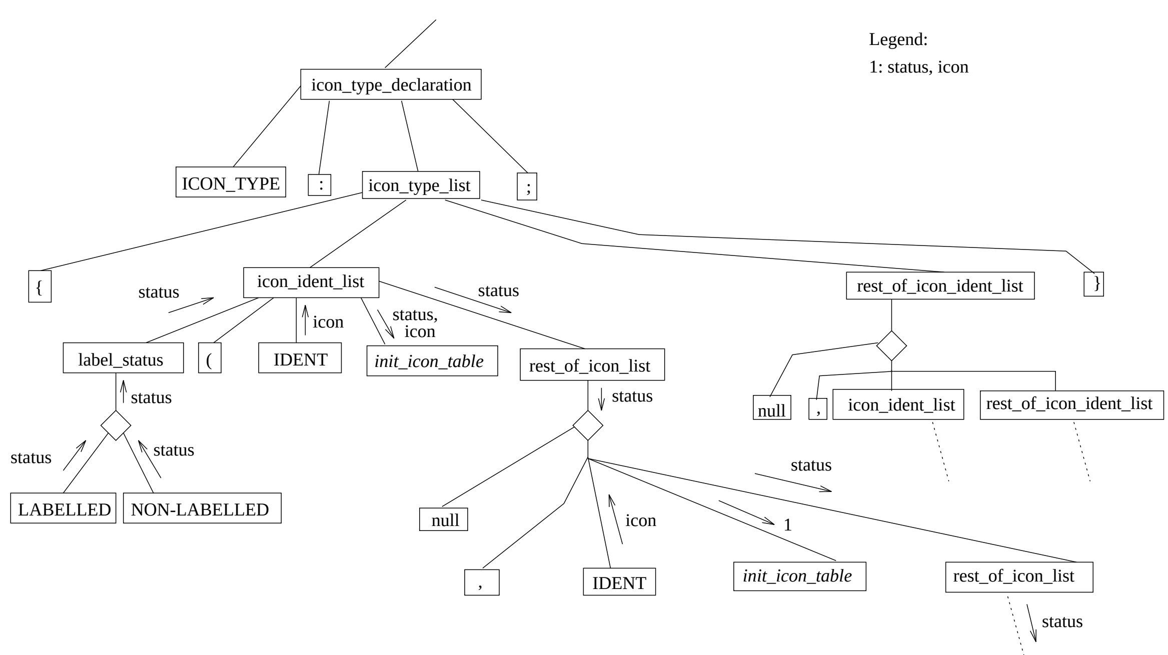


Figure 5.14: Icon Type Declaration Structure

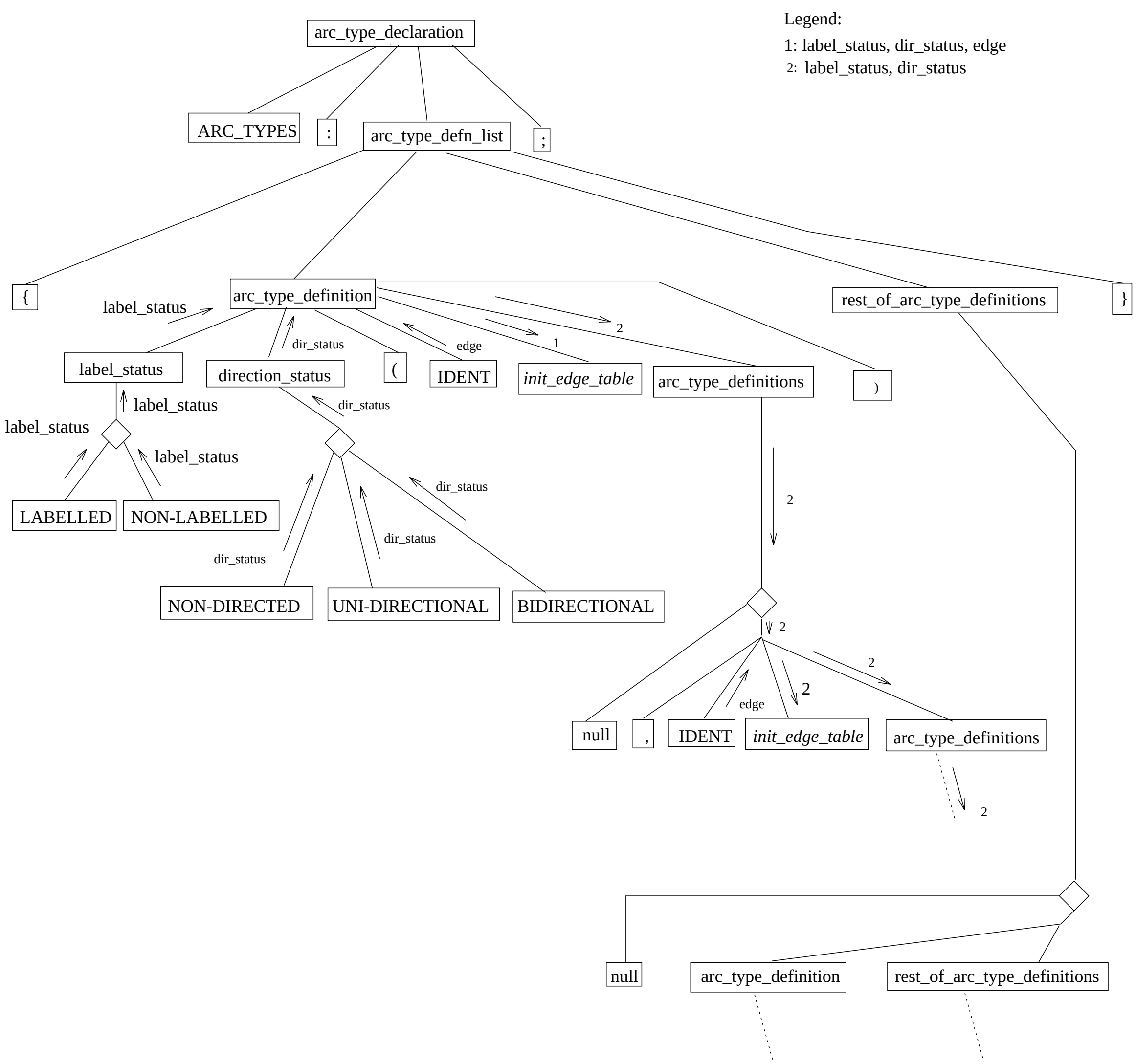


Figure 5.15: Edge Type Declaration Structure

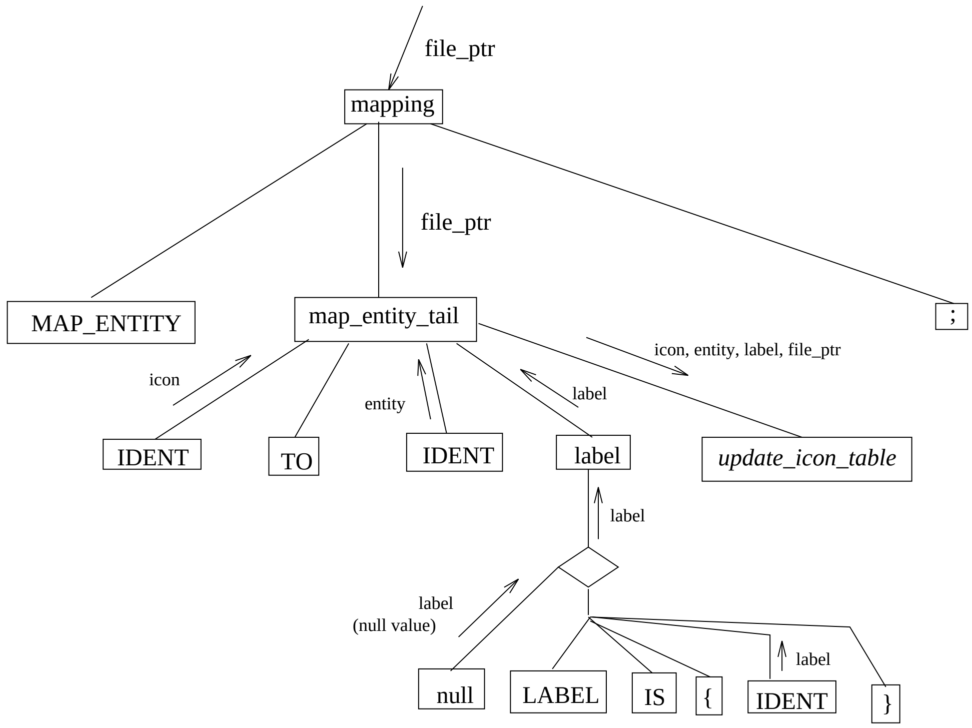


Figure 5.16: Icon Type-Entity Type Mapping Structure

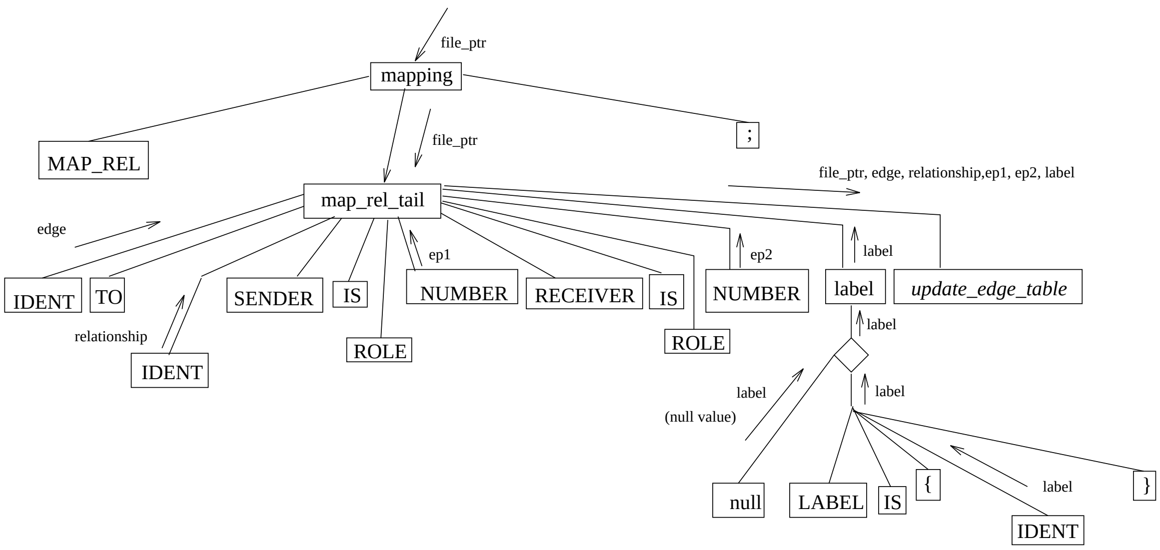


Figure 5.17: Edge Type-Relationship Type Mapping Structure

and the conflict resolution table. The action routines responsible for the generation of these tables are *update_icon_placement_table* for the icon placement table, *update_et* and *get_edge_types* for the conflicting edges table, and *init_con_res_tab* and *update_con_res_tab* for the conflict resolution table. The *update_icon_placement_table* action routine is used in the *CASE* clause (Figure 5.18) of the grammar. The *update_et*, *get_edge_types*, *init_con_res_tab*, and *update_con_res_tab* action routines are used in the *CONFLICT* clause of the grammar, as illustrated in Figures 5.19, 5.20, 5.21, and 5.22, respectively. Figures 5.20, 5.21, and 5.22 are continuations of Figure 5.19 for the *edge_ident_list*, *simple_mapping_expr* , and *simple_res_expr* non_terminal symbols, respectively.

update_icon_placement_table<edge, operator, entry1, entry2, entry3, entry4, file_ptr

Function: This routine fills in an icon placement table record. The *edge_type*, and *binary operator* fields of the icon placement table are filled in with the values of the *edge*, and *operator* attributes, respectively. The input attributes *entry1*, *entry2*, *entry3*, and *entry4* are indices into the symbol table stack for participant icon type one and two as retrieved in the *CASE* clause directly, and participant icon type one (sender) and two (receiver) as retrieved in the *icon_icon_or_group_placement* production. For example, if a *CASE* statement appeared in a program as follows:

```
CASE ent,pro:
    ABOVE(pro, ent);
ENDCASE
```

Then the *entry1*, and *entry2* indices would refer to the locations in the symbol table stack for the identifiers *ent* and *pro*, respectively (the CASE(ent,pro) line). The *entry3*, and *entry4* indices would refer to the locations in the symbol table stack for the identifiers *pro* and *ent*, respectively (the ABOVE(pro, ent) line). The *update_icon_placement_table* routine would then update the *participant one icon type* (sender) and *participant two icon type* (receiver) fields of the *icon placement table*

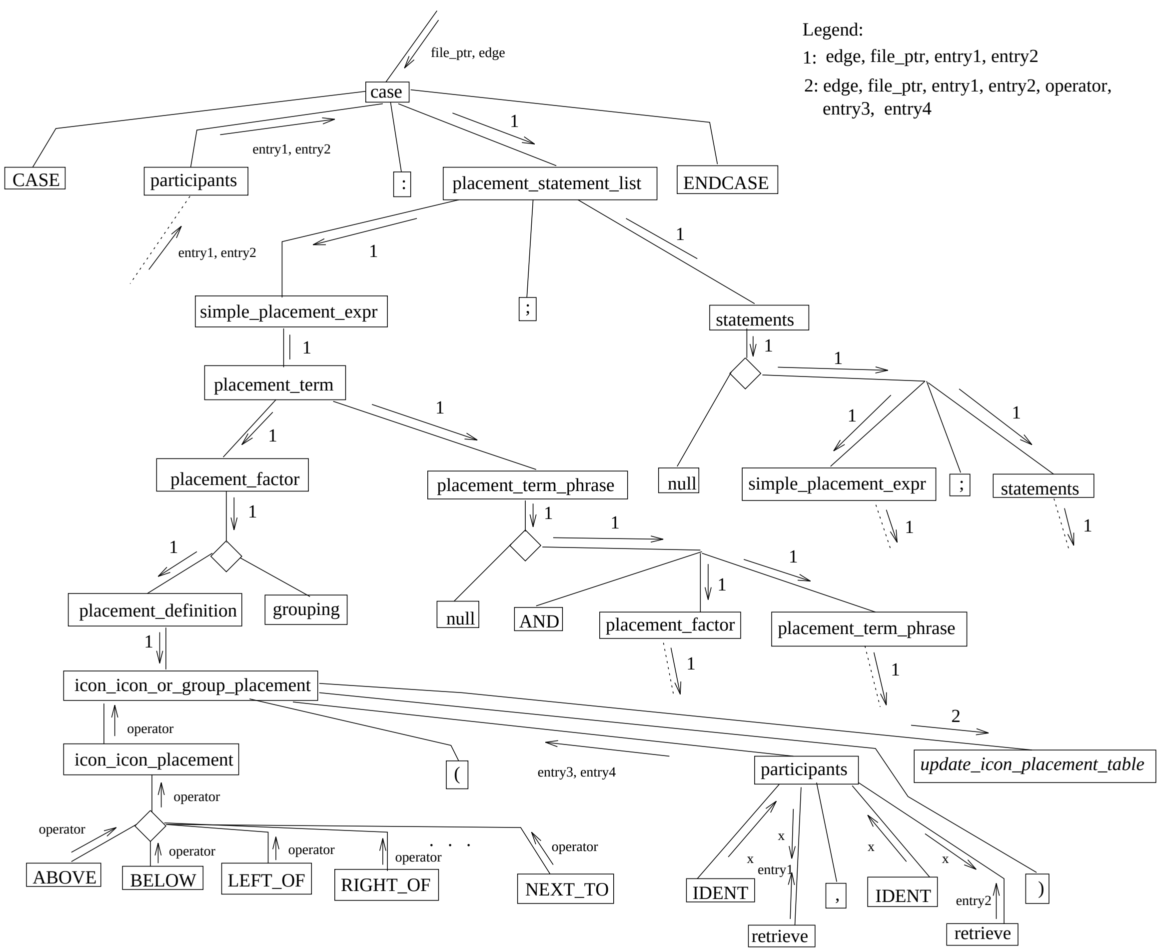


Figure 5.18: Icon Placement Structure

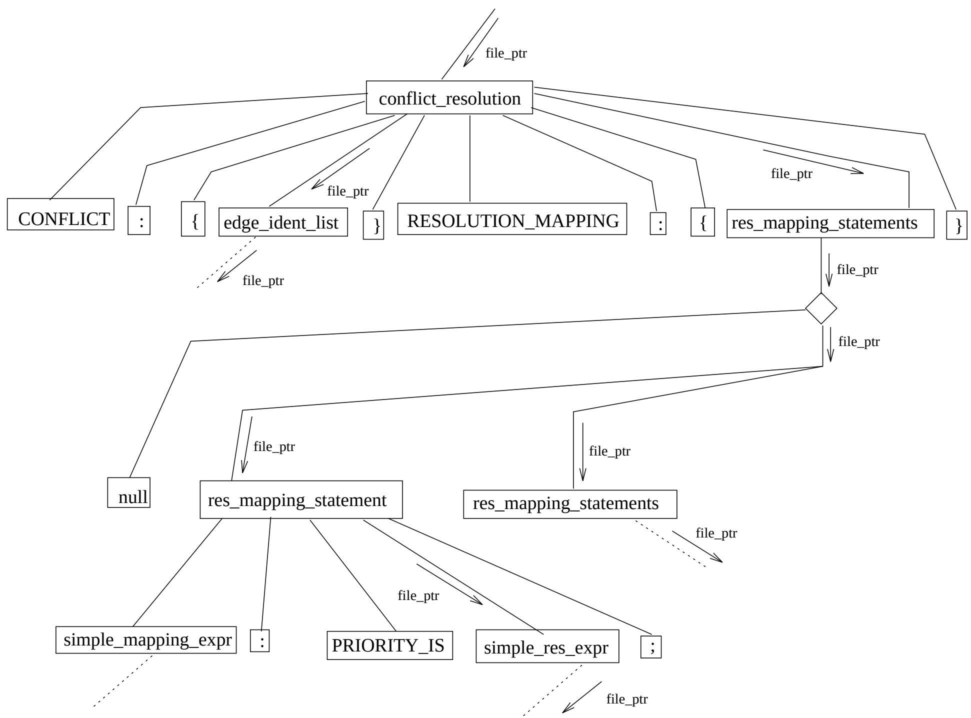


Figure 5.19: Conflict Resolution Structure

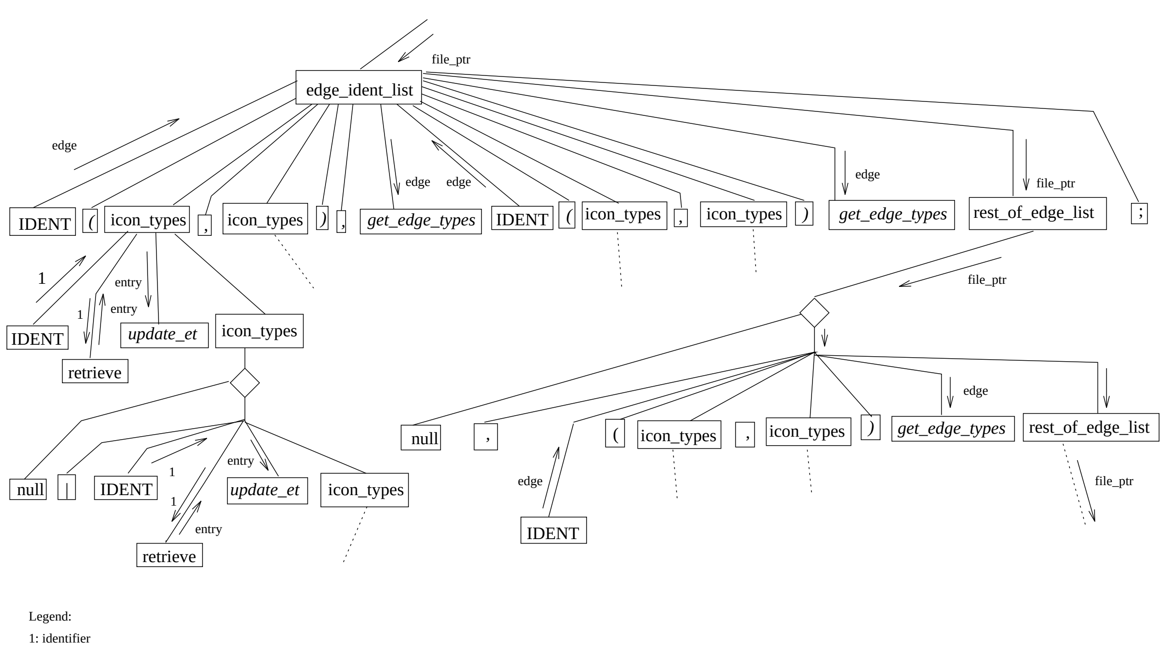


Figure 5.20: Conflicting Edges Structure

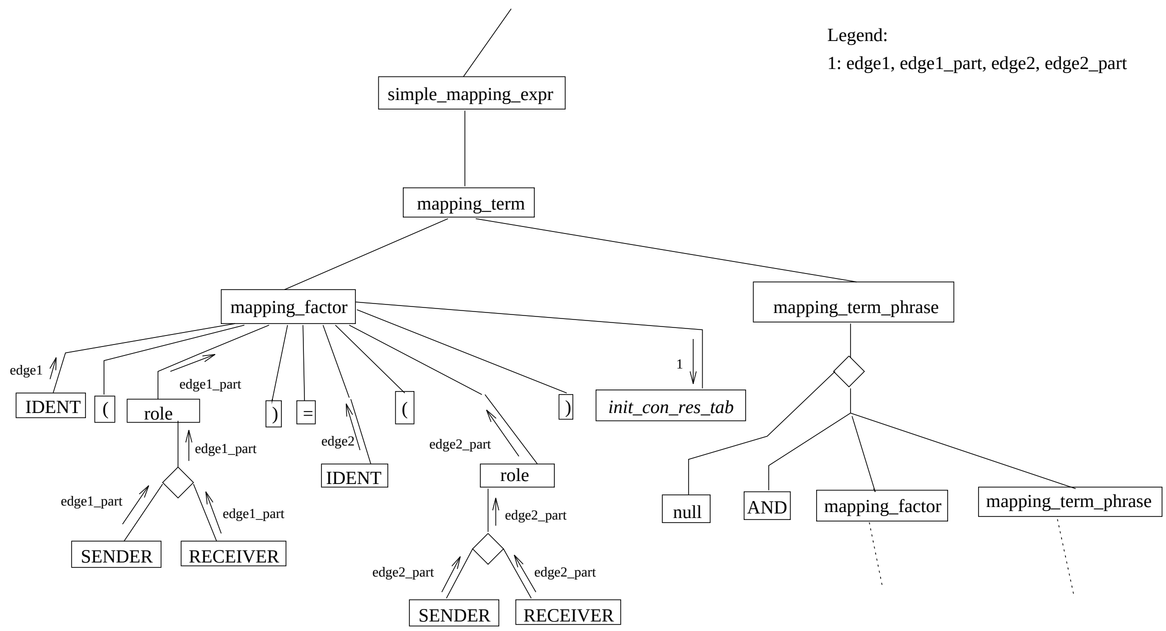


Figure 5.21: Conflict Condition Structure

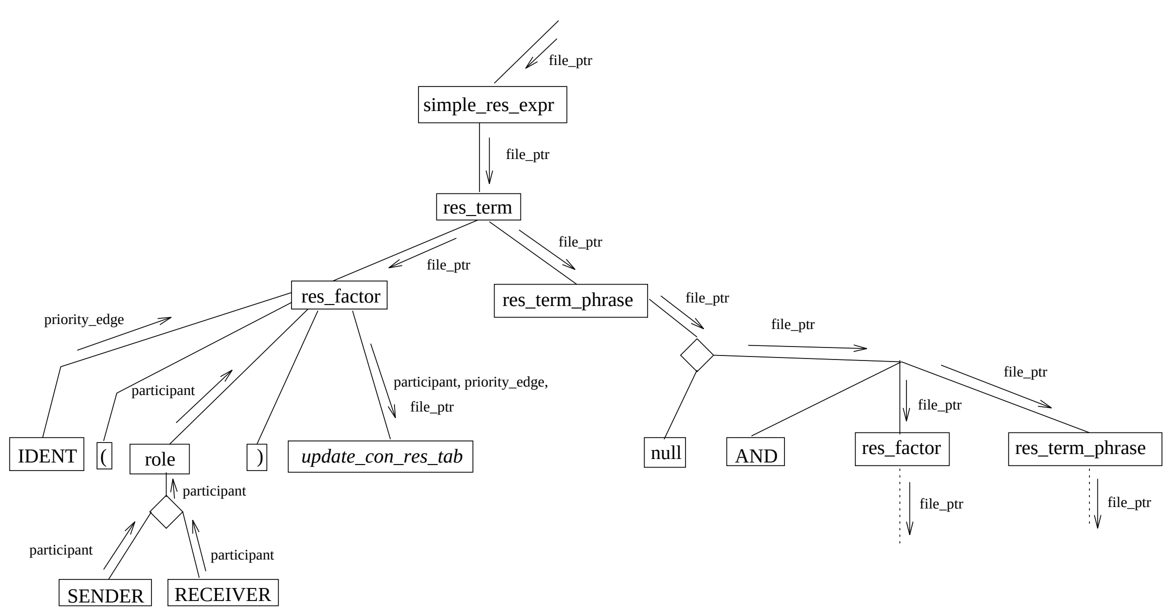


Figure 5.22: Conflict Solution Structure

Table 5.15: Sample *et* Array

process_icon	interface_icon
process_icon	

record. The routine would do so with the value of the *type name* field of the symbol table stack record at location *entry3*, and *entry4*. The *entry1*, and *entry2* values are inherited because it is necessary to verify that the identifiers mentioned in the placement statement, the *ABOVE(pro, ent)* line, are the same identifiers in the *CASE(ent,pro)* line. This record is then appended to the file pointed at by the *file_ptr* attribute.

update_et<entry

Function: The routine is responsible for updating the *et* array. This array contains all of the participating icon types for a given edge type listed in a *CONFLICT* clause of a program. The array is utilized by the action routine *get_edge_types*, which will be explained in the next paragraph. The *type name* field of the symbol table stack record at location *entry* of the stack is inserted into the array at the appropriate row, representing which role it is fulfilling (row one for sender and row two for receiver), and at the appropriate column, specifying which participant it is. For example, in Table 5.15 (repeat of Table 5.12), the sender participant group (row one) has two possible participating icon types, *process_icon* and *interface_icon*. The receiver participant group (row two) has one possible participating icon type, *process_icon*. The *get_edge_types* action routine clears the *et* array after it is finished with it, therefore creating a new *et* array for each edge type listed in the *CONFLICT* clause of a DLL program.

get_edge_types<edge,file_ptr

Function: This routine is responsible for updating the conflicting edges table. A record for the conflicting edge table is generated for all possible combinations of icon types listed in the *et* array for the current edge type listed in the *CONFLICT* clause of a DLL program. Therefore, for the *et* array of Table 5.12, two records would be

Table 5.16: Sample Conflicting Edges Table

uni_flow_sends	process_icon	process_icon
uni_flow_sends	interface_icon	process_icon

Table 5.17: Sample Conflict Resolution Table

uni_flow_sends	SENDER	uni_flow_sends	RECEIVER	uni_flow_sends	RECEIVER

generated. The *edge* attribute fills in the *edge type* field of the conflicting edges table, and the *participant one icon type* (sender) and *participant two icon types*(receiver) fields of the conflicting edges table are filled in with the icon types list in the *et* array. Table 5.16 illustrates what the conflicting edges table would look like assuming an edge type by the name of *uni_flow_sends* for the *et* array of Table 5.15.

init_con_res_tab<edge1, edge1_part, edge2, edge2_part

Function: This routine is responsible for partially filling a conflict resolution table record. The record is not written out to file at this stage. The *edge1*, *edge1_part*, *edge2*, and *edge2_part* attributes fill the *edge type 1*, *role for edge type 1*, *edge type 2*, and *role for edge type 2* fields of the conflict resolution table, respectively. These fields represent the condition for a conflict to occur in the placement constraints. For example, for the conflict resolution table of Table 5.17(repeat of Table 5.14), a conflict could occur if, for the given edge type *uni_flow_sends*, there exists instances of the edge type *uni_flow_sends* in which the sender participant of one instance equals the receiver participant of another instance. For example, Figure 5.23 illustrates a possible placement conflict situation for icons *A* and *B*. Assume that they are both the same type of icon. If a placement rule was generated that required the sender participant of the edge type *uni_flow_sends* to be placed above and to the left of the receiver participant, a direct placement conflict would exist between icons *A* and *B*.

update_con_res_tab<priority_edge, participant, file_ptr

Function: This routine is responsible for filling in the remaining fields, *priority edge type* and *priority role*, for the conflict resolution table. This routine scans through the conflict resolution table until the *priority_edge* and *participant* attributes match with either the first two or second two fields of the current record of the conflict resolution table. When a match is made, the record's *priority edge type* and *priority role* fields are filled with the values of the *priority_edge* and *participant* attributes, respectively. The updated record is then written out to the file pointed to by the *file_ptr* attribute.

5.3 The ADL Tool

This tool Tool provides the developer with the means of laying out Metaview specification databases according to the diagram layout rules encoded in the diagram modeling tables generated by the compiler which are described in Section 5.2.5.1.

The ADL Tool exists in two states, generic and configured, as explained in Section 4.7. The configured ADL Tool has all of the functionality of the generic ADL Tool, however, the former has diagram modeling tables associated with it. The developer is only required to know the name of the configured ADL Tool which corresponds to a particular diagram type. Since the configured tool is aware of what diagram modeling tables are to be used, the developer only provides the configured tool with the name of a specification database to be laid out, and the name of the output file where to write the object code.

First, a brief treatment of the intended object platform is given. This will be followed by a discussion of the ADL Tool's implementation.

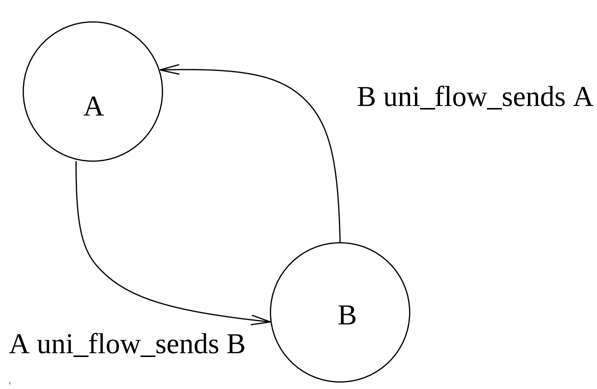


Figure 5.23: Conflict Condition For the Edge Type uni_flow_sends

5.3.1 The Object Platform — ICOLA

The configured ADL Tool generates code for the ICOLA platform. ICOLA (Incremental Constraint-based Object Layout Algorithm) is the layout engine of the WAND (not an acronym) visualization tool currently under development at the University of Saskatchewan. ICOLA displays pictures specified in the PDI (Picture Descriptions for ICOLA) language. PDI is a constraint-based, declarative picture description language which has graphical object declarations and constraint declarations. ICOLA produces images from a set of declared graphical objects, and constraints among these graphical objects. ICOLA is incremental in the sense that it does not require the entire set of object and constraint definitions at once. ICOLA can receive new object and constraint information piecemeal and generate a new image with each new piece of information that it receives without a complete recalculation after each new piece of information. ICOLA can also operate in a batch mode. This mode allows ICOLA to read in a large set of constraints without solving any of them until a *solve* command is given.

One of ICOLA's strength's is its simplicity. This was an important factor in choosing the ICOLA platform as the output target of the ADL system.

5.3.1.1 PDI: The ICOLA Input Language

The ICOLA platform accepts a very simple declarative language. PDI is a terse language which, for the purposes of this thesis, consists essentially of seven commands:

- *object(object_name, object_type)* — This command defines an instantiation of a predefined *object_type*.

- *constraint (constraint_name, list_of_objects)* — This command defines a specific relationship between instantiated objects. For example, *constraint(above, [a,b])* constrains object *a* to be *above* object *b*.

- *mode_batch* — This command instructs ICOLA to read in an entire PDI file before attempting to solve anything.

- *reset* — This command initializes ICOLA.
- *solve* — This command instructs ICOLA to attempt to solve all its input constraints.
- *group(group_name)* — This command instructs ICOLA to group all of the commands it encounters prior to an *end_group(group_name)* command. The grouped commands are then treated as a single object.
- *end_group(group_name)* — This command terminates a previous *group(group_name)* command.
- *attribute (object_name, attribute_name, attribute_value)* — This command allows a reassignment of values to any object's attributes. These attributes include, but are not limited to, color, label, and visibility.

For a more detailed explanation of the input language to ICOLA refer to Kusalik and Oster[15].

An example is given to illustrate the PDI language. The code of Figure 5.24 produces the simple picture of Figure 5.25. The PDI code *resets* ICOLA and puts ICOLA in batch mode. The code defines five *box* objects (*a, b, c, d,* and *e*), five *text* objects (*a_label, b_label, c_label, d_label,* and *e_label*), and three *polyline* objects (*l1, l2,* and *l3*). The polyline objects are the same type of object which is used by the ADL system to route edges on a diagram. The PDI code then constrains the text objects within the box objects to represesent labels. The box objects are then constrained to be relative to one another via PDI placement *constraint* commands. The code then constrains the polyline objects to connect up three pairs of the box objects. The connecting of the box objects with the polylines is accomplished by the ADL edge router.

The picture created by the PDI code illustrates the placement of the five box objects and their associated labels as well as the grid routing standard with the three routed polyline objects.

```
reset.
mode_batch.

object( a, box ).
object(a_label,text).
attribute(a_label,text,"a").
object( b, box ).
object(b_label,text).
attribute(b_label,text,"b").
object( c, box ).
object(c_label,text).
attribute(c_label,text,"c").
object( d, box ).
object(d_label,text).
attribute(d_label,text,"d").
object( e, box ).
object(e_label,text).
attribute(e_label,text,"e").
object(l1,polyline).
object(l2,polyline).
object(l3,polyline).

constraint(inside,[a_label,a]).
constraint(inside,[b_label,b]).
constraint(inside,[c_label,c]).
constraint(inside,[d_label,d]).
constraint(inside,[e_label,e]).
constraint( below, [ b, a ]).
constraint( right_of, [ b, a ]).
constraint( below, [ c, a ]).
constraint( right_of, [ d, a ]).
constraint( right_of, [ e, a ]).
constraint( below, [ e, b ]).
constraint( horizontal_distance, [150, a, e]).
constraint( vertical_distance, [70, a, e]).
constraint(connects,[l1,a,d]).
constraint(connects,[l2,d,b]).
constraint(connects,[l3,d,e]).
solve.
```

Figure 5.24: Sample PDI Code

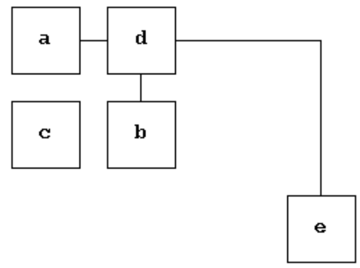


Figure 5.25: Sample Picture Generated From Sample PDI Code

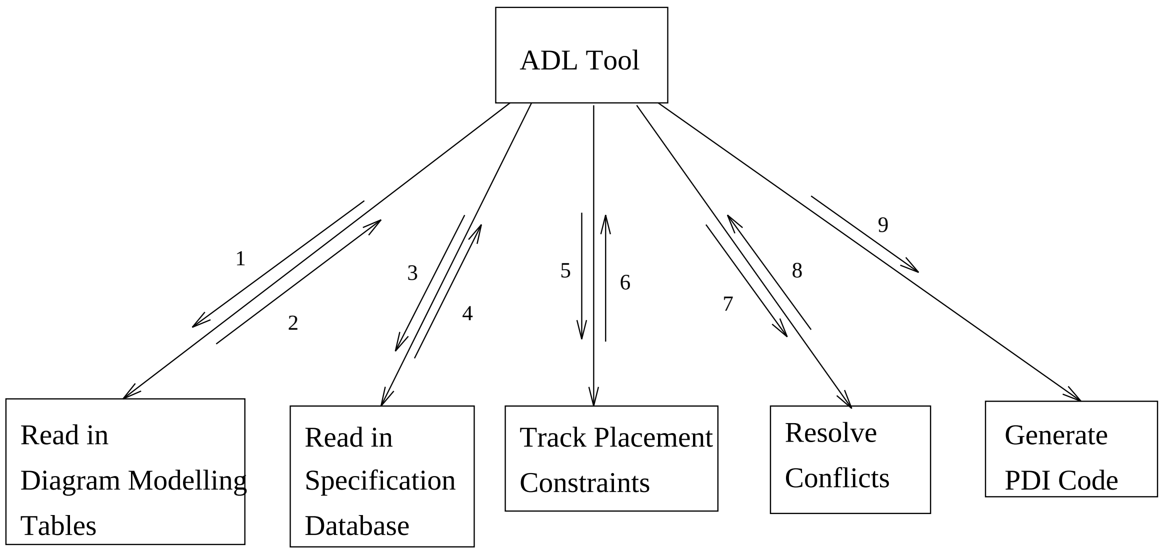
5.3.2 The ADL Tool Implementation

The main structure of the ADL Tool is illustrated in Figure 5.26. The main components of the generic ADL Tool are decomposed and discussed in this section. The ADL Tool is implemented in a modular form with each component being responsible for one task. There are five main components in the ADL Tool, as illustrated in the Yourdon[29] structure chart in Figure 5.26. The first component performs the task of reading the required diagram modeling tables into internal tables. The next three components use the in-core tables to read in and process a developer-named specification database. The last component writes out the ICOLA PDI object code to a developer-named output file. ICOLA can then read in this file and perform the actual layout, i.e., the placement and routing, of the diagram.

The generic ADL Tool is configured by a methods engineer as discussed in Section 4.7. The configured ADL Tool is a shellscript written around the generic ADL Tool with the name of the diagram modeling tables file hard-coded as input to the generic ADL Tool. The Configured ADL Tool then only requires the developer to specify the name of the input specification database and the output file for the PDI code.

5.3.3 The Data Structures

The ADL Tool uses the diagram modeling tables, therefore the six tables described in Section 5.2.5.1 form a large part of the data structures used by the tool. Beyond these six tables, however, the generic ADL Tool uses two other tables to record the specification database information and the icon placement information. These two



Legend:
1: name of diagram modeling tables file
2: diagram modeling tables
3: name of specification database, icon type and edge type tables
4: icon and edge instance tables
5: icon type placement table, icon and edge instance tables
6: icon instance table
7: icon and edge instance tables, conflicting edge types table, conflict resolution table
8: icon instance table
9: icon and edge instance tables, output file name

Figure 5.26: Structure Chart of the ADL Tool

tables are referred to as the *icon and edge instance tables* on Figure 5.26. The first table is called *ent_tab*, which is an abbreviation for *entities table*. Each record in this table represents one instantiated entity type. The second table is called *rel_tab*, which is an abbreviation for *relationships table*. Each record in this table represents one instantiated relationship type.

The structure for the *ent_tab* table is illustrated in Table 5.18. The *name* field corresponds to the Metaview attribute *name* of the instantiated entity type. The *type* field is the name of the Metaview entity type. The *label* field holds the value of the intended icon label. The *icon* field is the name of the corresponding icon type. The *label_status* field is a flag which dictates whether or not the given icon is to be labelled.

Table 5.18: *ent_tab* Table Structure

name	type	label	icon	label_status	left_of	right_of	above	below	···

label_status can have the values, *LABELED*, or *NON-LABELED*. The four remaining fields, *left_of*, *right_of*, *above*, *below*, etc. are one-dimensional arrays corresponding to the binary placement operators. These arrays hold the indices into the *ent_tab* table (this table) of entities/icons which are placed relative to the given entity/icon. For example, if the *above* array had two elements stored in it; 1 and 3, then this would mean that the entities at positions 1 and 3 in this table are to be placed *above* the given entity/icon.

The structure for the *rel_tab* is illustrated in Table 5.19. The *name* field corresponds to the ADL Tool-assigned name of the instantiated relationship type. Currently, instantiated relationship types are not uniquely identified by a *name* attribute within the specification database, but by the relationship type name and a list of participating entity type instances. Therefore, when the ADL Tool reads in an instantiated relationship type, it assigns a unique name to it for future reference. The *type* field is the Metaview relationship type name. The *edge* field is the edge type name. The *label* field is the label of the edge. The *label_status* is a flag which determines whether or not this edge is to be labelled. The *label_status* field can hold the values, *LABELED*, or *NON-LABELED*. The *direction_status* field is a flag which determines whether or not arrowheads will be applied to the edge. The *direction_status* field can hold the values, *NON-DIRECTED* (no arrowhead), *UNI-DIRECTIONAL*,(one arrowhead at the end of the edge which connects to the second primary participant; the receiver) and *BI-DIRECTIONAL* (one arrowhead at each end of the edge). The fields *participant1* and *participant2* contain the names of the sending and receiving entities of the relationship, respectively.

5.3.4 The Main Components

Recall that the ADL Tool is composed of five primary components, as illustrated in Figure 5.26. These components are further discussed and decomposed in this section.

Table 5.19: *rel_tab* Table Structure

name	type	edge	label	label_status	direction_status	participant1	participant2

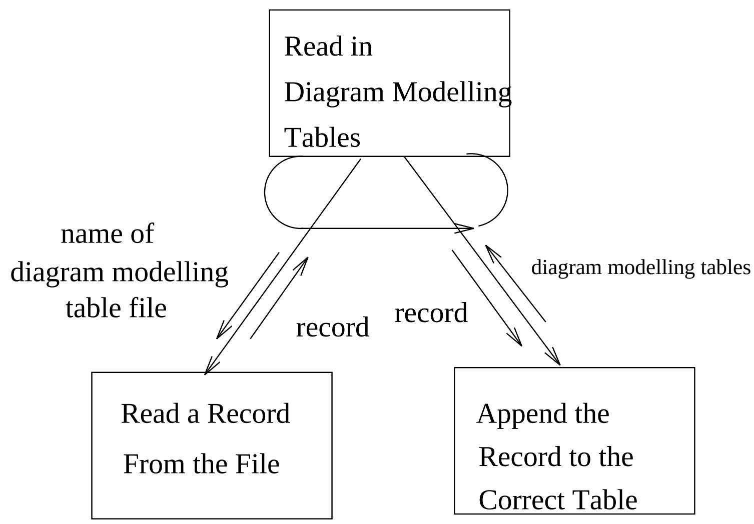


Figure 5.27: Reading in the Diagram Modeling Tables

5.3.4.1 Reading in the Diagram Modeling Tables

This component is responsible for reading the appropriate diagram modeling tables from disk into internal tables. The diagram modeling tables are described in Section 5.2.5.1. All diagram modeling tables are stored in one file. Each record of that file has a table identification number to allow this routine to determine which internal table a given file record should be appended to. This task is illustrated in Figure 5.27.

5.3.4.2 Read in Specification Database

This component is responsible for reading a specification database into the *ent_tab* and *rel_tab* internal instance tables. These tables are discussed in Section 5.3.3. This component uses the Icon and Edge Diagram Modeling Tables, and partially fills the *ent_tab* and *rel_tab* tables. This component directly interfaces with the Metaview project daemon. The Metaview project Daemon was briefly discussed in Section 2.3. Figure 5.28 illustrates the decomposition of the current component. A high level pseudo-code algorithm of this component is given as follows.

1. for each record in the icon table do
2. get a list of entities from the Metaview project daemon

3. for each entity in the list of entities do
4. put the appropriate information into a ent_tab table record
5. for each record in the edge table do
6. get a list of relationships from the Metaview project daemon
7. for each relationship in the list of relationships do
8. put the appropriate information into a rel_tab table record

The algorithm is explained in more detail in the following paragraphs. For each record in the icon table, a query is sent to the Metaview project daemon. The query asks the project daemon to provide a list of all instances of an entity type specified by the field *entity type* of the current icon table's record. The returned list is parsed and an *ent_tab* table record is appended to the *ent_tab* table for each instance of the entity type listed. The *ent_tab* table's *name* field is filled in by the name attribute of the current entity type instance. The icon table record's fields, *entity_type*, *icon_type*, and *label_status* are inserted into each *ent_tab* record. These fields are inserted into the *type*, *icon*, and *label status* fields of the *ent_tab* table, respectively. The value of the icon table record's *label_source* field determines what value will be used for the *ent_tab* table record's *label* field. The remaining *ent_tab* fields, *left_of*, *right_of*, *above* and *below* are filled in by the track placement constraints component of the ADL Tool.

Similarly, for each record in the edge table, a query is sent to the Metaview project daemon. The query asks the project daemon to provide a list of all instances of a relationship type specified by the field *relationship type* of the current edge table's record. The returned list is parsed and a *rel_tab* table record is appended to the *rel_tab* table for each instance of the relationship type listed. A unique name is assigned to each relationship by this ADL Tool component because Metaview, currently, does not provide unique names for relationships like it does for entities. The *name* field of the *rel_tab* table record is filled in with this assigned name. The edge table record's fields, *relationship type*, *edge_type*, *label_status*, and *direction_status* fields are inserted into

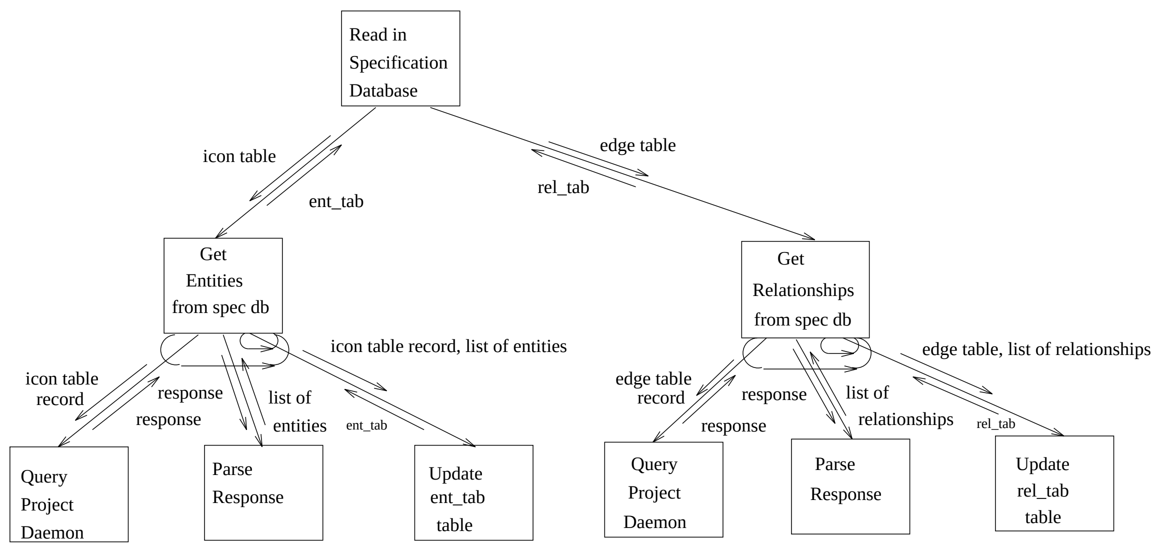


Figure 5.28: Read in the Specification Database

each *rel_tab* table record. These fields are inserted into the *type*, *edge*, *label_status*, and *direction_status* fields of the *rel_tab* table, respectively. The sender and receiver participant entity names fill in the fields *participant1* and *participant2* fields of *rel_tab* respectively.

5.3.4.3 Track Placement Constraints

This component is responsible for updating the *above*, *below*, *left_of*, *right_of*, etc. icon placement array fields for each record of the *ent_tab* table. This process is illustrated in Figure 5.29. A high-level pseudo-code algorithm for this component follows.

1. reset the icon placement arrays for each entity in the ent_tab table
2. for each record in the icon placement table do
3.   for each record in the rel_tab table do
4.   if the edge type of the icon placement table record matches
            the edge type of the current relationship then
5.   put the current relationship into a relavant relationship list
6.   for each record in the relevant relationship list do
7.   update the placement arrays for the sender and receiver entities
            of the current relationship

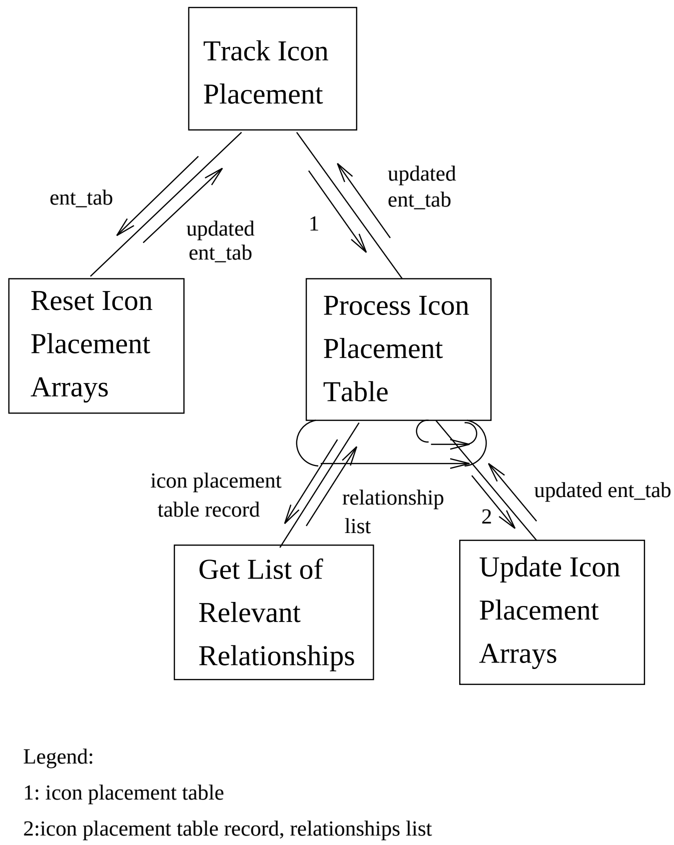


Figure 5.29: Track Placement Constraints

The algorithm is explained in more detail in the following paragraphs. The first task performed is to clear all of the placement array fields. The next step is to iterate through the icon placement table. For each icon placement table record, a list of relevant relationships from the *rel_tab* table is acquired. The criteria for a relationship entry from the *rel_tab* table to make it into this relationships list is as follows:

- The *edge type* field in the given icon placement table record matches with the *edge* field of the current *rel_tab* record.

- Both *participant one icon type* (sender) and *participant two icon type* (receiver) fields of the *icon placement table* match with the icon types of the *participant1* and *participant2* fields of the *rel_tab* table.

Once the relationships list for the given icon placement table record is composed, the list is then traversed once. For each relationship in the list, the *participant2* field's placement array corresponding to the current icon placement table record's binary operator, *above*, *below*, etc., is updated with the *participant1* field. This procedure is continued until all records in the icon placement table have been processed.

For example, a relationship in the list of relevant relationships had entity *book* as the value for *participant2* and the entity *category* as the value for *participant1*. The current icon placement record's *binary placement operator*'s field is *ABOVE*. This would mean that the *above* array field, of the record in the *ent_tab* corresponding to the entity *book*, would have the entity *category* appended to it to represent that there is a placement constraint between the two entities *book* and *category*.

5.3.4.4 Resolve Conflicts

This component is responsible for finding and resolving conflicts within the placement arrays of the *ent_tab* table. The high-level pseudo-code of this component is given below.

```
1. for each record in the conflicting edges table do
2.       for each record in the rel_tab table do
3.            if the edge type of the conflicting edges table record matches
                   the edge type of the current relationship then
4.                 put the current relationship into a relavant relationship list
5. for each record in the conflict resolution table do
6.       for each record in the relevant relationships list do
7.            if the edge type 1 field matches with the edge type field of the
                   current relationship then
8.                 for each record in the relevant relationships list do
9.                      if the edge type 2 field matches with the edge type field of the
                             current relationship then
10.                          if the sender of the first relationship matches the receiver
```

of the second relationship and vica-vera then

11. remove the non-priority icon from the priority icon's placement arrays in the ent_tab table

The algorithm is explained in more detail in the following paragraphs. The conflicting edges table is used to get a list of relevant relationships similar to the way that this list is arrived at in Section 5.3.4.3. This component, however, bases the relationships list on the entire conflicting edges table, and not just on a single record.

The conflict resolution table is then traversed once. For each record in the conflict resolution table, the relevant relationships list is traversed until the *edge* field in a record of the relationships list matches the *edge type 1* field of the conflict resolution table. The relationships list is then traversed again to match the *edge* field to the *edge type 2* field of the conflict resolution table. This process is $O(n^2)$. For a direct placement conflict to exist, the sending and receiving entities of the relationships must match according to the criteria in the conflict resolution table record. However, for a conflict to occur, the sender participant of the first relationship must equal the receiver participant of the second relationship, and, of course, vice versa. This is the only way a direct placement conflict can occur.

Once these conflicting relationships are found, then the direct placement conflict is resolved in favor of the relationship whose corresponding *edge* field matches the *priority edge type* field of the given conflict resolution table record. In the case where the conflicting relationships correspond to the same edge type, then the first relationship found is considered the priority relationship. Also, the priority relationship's priority entity's placement tables are updated to remove the non-priority entity. The direct placement conflict is, therefore, removed.

5.3.4.5 Generate PDI Code

This component is responsible for generating all of the ICOLA PDI object code to the output file. Figure 5.31 illustrates the main tasks of this component. All output

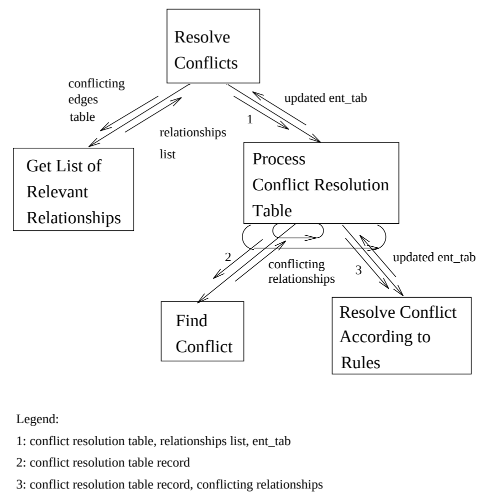


Figure 5.30: Resolve Conflicts

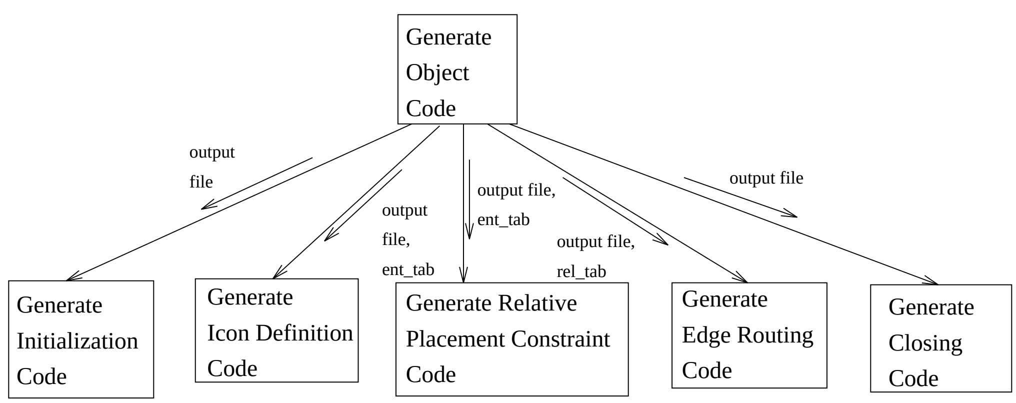


Figure 5.31: Generate Object Code

files have header PDI code which sets the ICOLA platform into the correct mode. For example, the following code is written at the top of every output file:

```
reset.
mode_batch.
```

This code resets ICOLA for a new diagram, and puts it into *batch mode*. Batch mode means that ICOLA postpones solving all constraints until a *solve* command is received.

The icon definition task traverses through the *ent_tab* table. It reads in the correct icon definition, according to the *icon* field of the current *ent_tab* table record, from the icon library. The icon definition is massaged to adapt it to the specific entity. If the *label_status* field of the current *ent_tab* table record has the value *LABELED*, then the value of the *label* field (of *ent_tab*) is inserted into the icon definition. The label is split up, and put on separate rows if it is too long. Essentially, if the label has "_" characters present then these are where the breaks are made in the label. The "_" characters are not displayed with the label on the icon. For the Metaview example of Section 4.2.2, the *book* entity corresponds to the *entity_icon* icon type, as shown in Section 4.5.2. The *entity_icon* exists in the icon library in the following form:

```
group(entity_icon).
    object( entity, box ).
    group( label0 ).
        object(label1,text).
        attribute(label1,text,"label").
    end_group( label0 ).
    constraint( below,[label0,top(entity)] ).
    constraint( right_of,[label0,left(entity)] ).
    constraint( left_of,[label0,right(entity)] ).
    constraint(horizontal_distance,[100, left(entity),right(entity)]).
    constraint(vertical_distance,[75, top(entity),bottom(entity)]).
end_group(entity_icon).
```

It is transformed into the following form by the icon definition task:

```
group(book).
    object( entity, box ).
    group( label0 ).
        object(label1,text).
        attribute(label1,text,"book").
    end_group( label0 ).
    constraint( below,[label0,top(entity)] ).
```

```
    constraint( right_of,[label0,left(entity)] ).
    constraint( left_of,[label0,right(entity)] ).
    constraint(horizontal_distance,[100, left(entity),right(entity)]).
    constraint(vertical_distance,[75, top(entity),bottom(entity)]).
end_group(book).
```

The group name was modified to be the name of the entity. The label was modified to reflect the value of the *label* field in the current *ent_tab* table record. All icon definitions are similar to the one presented here.

The next task is responsible for generating the placement constraint code. This task traverses the *ent_tab* table and generates the appropriate ICOLA constraints for each of the *above*, *below*, *left_of*, *right_of* arrays in the current *ent_tab* table record. Essentially, a placement constraint is generated for each entry in each of the four arrays. These constraints are then written to the output file. For example, consider two entities *book* and *title*. If the entity *title* is in the *book* entity's *above* field array then the following placement constraint code would be emitted to reflect that.

```
constraint(above,[title, book]).
```

Similar placement constraints are generated for all *ent_tab* table's placement array entries.

The next step involves generating the PDI code which defines edges, specifies them to be routed, labels them (if necessary) and adds an arrowhead(s) (if necessary). The *rel_tab* table is traversed and all relationships listed have the appropriate PDI code generated. This code defines the edge, and specifies it to be routed (according to the edge routing algorithm of Section 5.4). If labels are required (the *label_status* field for the current relationship has the value *LABELED*) then code which places the value of the *label* field at a default location is generated. If a single arrowhead is required (the *direction_status* field for the current relationship has the value *UNI_DIRECTIONAL*), then the PDI code which causes the routed edge to have an arrowhead will be generated. The arrowhead will be placed on the end of the routed edge where the receiving entity is connected. If the *direction_status* field for the current relationship has the value *BI_DIRECTIONAL*, then an arrowhead would be placed at both ends of the routed edge.

For example, assume that the *book* entity is the sending entity and the *title* entity is the receiving entity of a relationship listed in a given *rel_tab* table record. The following code would be generated.

```
object(edge0,polyline).
constraint(connects,[edge0,book,title]).
```

This PDI code would cause ICOLA to invoke the A* edge routing algorithm to route the edge *edge0* from the *book* icon to the *title* icon. This particular edge is unlabeled and undirected, hence the lack of PDI code to place a label and an arrowhead on it. Assume, now, that this edge was required to have a label and a *UNI_DIRECTIONAL* arrowhead. The following code would then be generated.

```
object(edge0,polyline).
constraint(connects,[edge0,book,title]).
attribute(edge0,arrowhead,head).
attribute(edge0,[["has"]]).
attribute(edge0,label_pos,[right]).
```

The last step in code generation is to write the PDI command *solve* to the output file. This command instructs ICOLA to begin solving all of the placement and routing constraints.

5.4 The A* Edge Routing Algorithm

The routing of the edges encoded in the output file of the ADL Tool is handled by the A* router. The A* router is explained in detail in this section along with the interface to the ICOLA platform. The A* Edge routing algorithm is implemented within the ICOLA object platform, as illustrated in Figure 4.1. The A* router is called by ICOLA after ICOLA has resolved all of the placement constraints. The A* router then proceeds to attempt to find appropriate paths for all of the edges encoded in the ADL Tool output file as described in Section 5.3.4.5. First, an explanation of the data structures used by the A* router will be provided, followed by the detailed pseudo-code of the algorithm. A sample trace of the algorithm will conclude this section.

5.4.1 Primary Data Structures in the A* Algorithm

The following data structures are used in the A* algorithm. The *Grid* is an array of cells or nodes which corresponds to the two-dimensional plane in which ICOLA operates. Each cell corresponds to one *unit* or *pixel* in the ICOLA plane. The Grid has all of the icons placed within it. The edge routing algorithm uses this grid as the search space to find an appropriate path between sets of two pre-placed icons for each edge to be routed. *StartCell* is the start node, *TargetCell* is the goal node. *OpenQueue* is an ordered list of nodes (ordered by lowest priority value; see below), also known as an abstract data type called a priority queue. The operations allowed on the *OpenQueue* ADT are:

- *PQInsert(Q,element,priority)* — insert *element* into *Q* with the priority value *priority*;
- *PQDelete(Q)* — delete the first element from *Q* and return with its value;
- *PQEmpty(Q)* — returns *true* if *Q* has no elements, and returns a *false* if there is at least one element in *Q*.

ClosedSet is a set of previously-searched nodes (order doesn't matter). In general, nodes that need to be searched are put on *OpenQueue* (at the proper position). As they are searched, they are removed from *OpenQueue* and put in *ClosedSet*. Occasionally a newer, better route will be found to a node after it has already been searched, in which case we remove it from *ClosedSet* and put it back on *OpenQueue* to be reconsidered. *G[current]* is the distance already traveled to get from *StartCell* to node *current*, and is known exactly. *H(current)* is a function (the heuristic) which returns an estimate of the distance from node *current* to *TargetCell*. *F[current]* is the estimated distance from *StartCell* to *TargetCell* by going through node *current*, and is computed by *F[current] = G[current] + H(current)*. *H(current)* can be calculated for any node, but *F[current]* and *G[current]* only become defined when node *current* is visited.

PredCell ("Predecessor Cell") is defined for each node, and is a list of "came from" indications, so when *TargetCell* is finally reached, the *PredCell* structure is traversed to construct a path back to *StartCell*. The *PredCell* data structure is implemented as a simple two-dimensional vector in which each element of the vector contains an ordered pair of numbers which represent coordinates to another element of the *PredCell* vector. The algorithm would start with *PredCell[TargetCell]*, which is the cell coordinate of the cell adjacent to the *TargetCell* which will be used as part of the routed path. The *PredCell* value for *PredCell[TargetCell]* would then be used to find the next path cell until *StartCell's* coordinates are eventually reached. Therefore, traversing from *TargetCell* back to *StartCell* via the *PredCell* data structure is basically a singly-linked list traversal. Note that in Figure 5.32 the arrows on the diagram represent the *PredCell* data structure.

The *PathCoords* data structure is a one-dimensional array which contains a list of coordinates on the grid which coorespond to the *TargetCell, StartCell*, and the coordinates of all of the bends that the routed edge has. This array is returned to ICOLA to be drawn on the diagram.

5.4.2 Detailed Pseudo-code of the A* Algorithm

The A* algorithm algorithm is a relatively simple search mechanism which has been applied to the problem of edge routing for diagram layout. The A* algorithm follows. The algorithm uses the shortest path to completion as the heuristic to determine the priority of the node to be enqueued onto the *OpenQueue*.

A* Algorithm: (* heuristic search *)

Declarations:

Grid - the main routing search space

Edge - The ICOLA polyline object which represents a relationship

StartCell, TargetCell - The source and destination cells for the search

OpenQueue - The main priority queue for the search

PredCell - The predecessor cell array which is used to get the coordinates
  of the bends, to return to ICOLA, in a successfully routed edge

NeighborSet - This is the set of the four immediate neighbors
  of the current node being searched

neighbor - One of the neighbor nodes in NeighborSet

current - This is the current node being searched

Counter - This is a variable used to assist in the traversal of PredCell

PathCoords - This is an array which holds the coordinates of all of
  bends for the current routed edge

CurrentPredCell, PrevPredCell - These are used to assist in the traversal of
  PredCell

[Step 1. initialize the data structures]

  InitICOLAGrid(Grid)

  For each Edge DO

    GetStartCellandTargetCell(Grid, Edge, StartCell, TargetCell)

    PQInsert(OpenQueue,StartCell,1) (* Use 1 as an initial priority. *)

    ClosedSet $\leftarrow \{\}$ (* the empty set *)

    G[StartCell] $\leftarrow$ 0, F[StartCell] $\leftarrow$ 0, PredCell[StartCell] $\leftarrow$ NULL,

    found $\leftarrow$ FALSE

[Step 2. perform phase 1 — heuristic search]

    WHILE not PQEmpty(OpenQueue) and not found

        and accumulated search time $<$ TimeOutLimit DO

```
current ← PQDelete(OpenQueue) (* node with smallest F value; lowest priority *)
ClosedSet ← ClosedSet ∪ {current} (* put current in Closed *)
IF current = TargetCell THEN found ← TRUE (* we're done *)
ELSE (* continue search through node current *)
    NeighborSet ← {set of neighboring nodes of current}
    FOR each neighbor in NeighborSet DO
        IF (neighbor is not on OpenQueue or in ClosedSet) or (neighbor is
        not blocked) THEN
            G[neighbor] ← G[current] + Distance(current,neighbor)
            (* estimate solution path length *)
            F[neighbor] ← G[neighbor] + H(neighbor)
            PredCell[neighbor] ← current (* remember where we came from *)
            (* put neighbor on OpenQueue according to its priority *)
            PQInsert(OpenQueue,neighbor,F[neighbor])
        ELSE (* neighbor is on OpenQueue or in ClosedSet or blocked *)
            IF neighbor is not blocked THEN
                IF (G[current] + Distance(current,neighbor)) < G[neighbor] THEN
                (* we've found a better route to neighbor *)
                    G[neighbor] ← G[current] + Distance(current,neighbor)
                    F[neighbor] ← G[neighbor] + H(neighbor)
                    PredCell[neighbor] ← current (* remember where we came from *)
                    IF neighbor is on OpenQueue THEN
                        (* reposition neighbor in OpenQueue according to F[neighbor] *)
                        search for neighbor in OpenQueue
                        pop neighbor off of OpenQueue
                        PQInsert(Q,neighbor,F[neighbor])
                    ELSE (* neighbor is in ClosedSet *)
                        (* remove neighbor from ClosedSet *)
                        ClosedSet ← ClosedSet - {neighbor}
```

```
                    PQInsert(OpenQueue,neighbor,F[neighbor])
[Step 3. perform phase 2 — retrace path from Targetcell back to StartCell]
    IF found THEN
      (* Traverse the PredCell data structure. *)
      Counter ← 1
      PathCoords[Counter] ← Targetcell
      CurrentPredCell ← PredCell[Targetcell]
      UpdateGridWithPath[Grid,CurrentPredCell]
      WHILE CurrentPredCell ≠ StartCell DO
        PrevPredCell ← CurrentPredCell
        CurrentPredCell ← PredCell[CurrentPredCell]
        UpdateGridWithPath[Grid,CurrentPredCell]
        IF CurrentPredCell is not in the same direction as
          PredCell[CurrentPredCell]
          Counter ← Counter + 1
          PathCoords[Counter] ← PrevPredCell
      Counter ← Counter + 1
      PathCoords[Counter] ← StartCell
      SendCoordsToICOLA( PathCoords )
    ELSE
      Inform user that Targetcell cannot be reached from StartCell
```

In the first step of the algorithm, ICOLA is interfaced via the *InitICOLAGrid* routine. This routine is responsible for allocating space for the search grid, and placing all of the icons in their correct locations on the grid so that the edge router can route around them. The placement of the icons on the grid involves the placement of markers in the cells which make up the area of the icon with *block* flags. These block flags are used by the algorithm when it scrutinizes neighbor cells to determine if the cell should be placed on the *OpenQueue* or not.

The A* algorithm then performs a heuristic search each edge to route. The *GetStartCellandTargetCell* routine determines a reasonable starting and ending point for the search for the current edge. The other data structures (*OpenQueue, ClosedSet, G, F,* and *PredCell*) are initialized. The next step of the algorithm involves performing the heuristic search for the current edge. In this step, cells are searched, one by one, until the *TargetCell* is reached, the *OpenQueue* is empty, or the search time for the current edge exceeds a pre-defined time limit (currently set to 20 seconds).

If the search was unsuccessful, an appropriate error message is generated and ICOLA doesn't attempt to draw the edge on the diagram. If the search was successful, then traverse the *PredCell* array in order to determine the edge's path. The traversal involves tracing back from the *TargetCell* to the *StartCell*, and noting any time a right-angle turn in the path is made. The coordinates for these right-angle turns, along with the *TargetCell* and *StartCell* coordinates, are placed in the *PathCoords* array. This array is given to ICOLA, via the *SendCoordsToICOLA* routine, in order to draw the edge. While the path is being traversed, each grid array cell along the path is updated with a special flag so searches for future edge paths know that an edge has been placed here. This flag is used in a similar manner as the *block* flags for the icons. Both of these flags are used to determine if the next neighbor cell should be classified as *blocked* and, therefore, not enqueued onto the *OpenQueue*. The criteria for a neigbor cell to be blocked were outlined in Section 4.8. The criteria are as follows:

- All routed edges should be a reasonable distance away from icons and should NOT pass through any icons.
- All routed edges should not be routed too close to other edges and should not be routed, either partially or entirely, on another routed edge's path.
- An edge may cross another edge.

5.4.3 Sample Trace

The behavior of the A* search algorithm is shown in Figure 5.32. The A* algorithm does not specify whether new cells are placed in front of or behind cells already on the open queue which evaluate to identical estimated path lengths. We will use the convention that they are placed in front of cells of the same distance. This will minimize the amount of time to insert a cell on the open queue.

In Figure 5.32(a), the source cell (r5c5) has been searched, and its four neighbors have been placed on the open queue. Each cell on the open queue contains, in parentheses, the estimated length of the shortest path from *S (StartCell)* to *T (TargetCell)* that goes through that cell. This estimation is based on the Manhattan distance between the current cell position and the target, *T*. After the first cell on the open queue (*r5c6*) has been searched and moved to the closed set, the configuration in Figure 5.32(b) is reached, where there are five cells on the open queue. Once the next cell (*r5c6*) has been searched, the configuration in Figure 5.32(c) is reached. Now the target cell (*r7c7*) is very close to being searched, and a solution will be found on the next iteration of the loop. When *r7c7* is searched, it will be recognized as the target cell, and the *PredCell* data structure will be used to construct a trace back to the source cell; refer to the bold arrows of Figure 5.32(c).

You can see that the search progresses more directly toward the target cell. The search is drawn toward the target. If we double the size of the problem, the search will search roughly twice as many cells, and if we triple the size of the problem, the search will search roughly three times as many cells. This linear, O(n), behavior makes the A* algorithm more attractive for autorouting than the quadratic Lee's algorithm. With the incorporation of the heuristic (the rule which guides the search in the direction most likely to succeed), it is more difficult to specify a worst case behavior. However, A* will never take more time than Lee's algorithm.

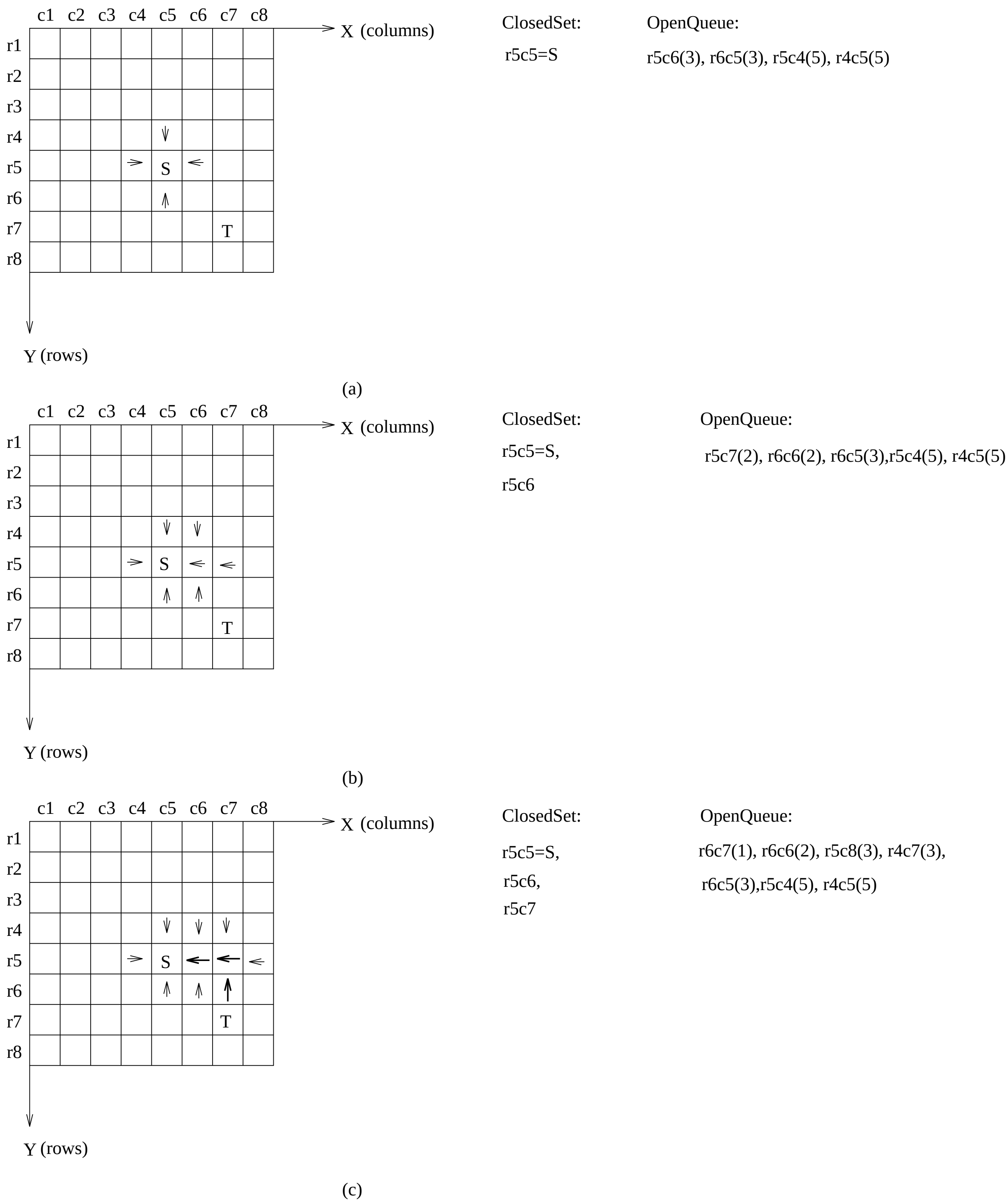


Figure 5.32: A Sample Run of the A* Algorithm

5.4.4 Time Complexity Analysis

To analyze this algorithm in the average to best case scenario, the active operations must be determined. There are three steps to the A* algorithm. The first and the third have O(1) and O(n) active operations, respectively. This can be seen by mere observation. The second step of the algorithm, however, has a more complex loop. The number of loop iterations, and therefore active operations, depends on how quickly the heuristic function is able to lock into the correct direction. This means that the cells which will eventually lead the search into the correct direction will be closer to the head of the priority queue, *OpenQueue*, whereas the cells that lead the search in the incorrect direction will be closer to the tail. The heuristic's function is to push the search in the correct direction while hopefully pruning the search space of bad solutions and keeping the good solutions still in the space to be searched. If we assume that the heuristic is pushing the search in the correct general direction then the number of cells to be searched, and thus the number of loop iterations, will be proportional to the number of cells which approximately separates the current cell's position and the target cell. This number of cells is approximately the Manhattan distance, n, between the two points. Thus, in the best case, the second step of the algorithm will run in O(n) time. Therefore, the A* algorithm runs in O(n) time.

5.5 ADL System Implementation Summary

The DLL compiler accepts a DLL program as input. The DLL compiler writes the diagram modeling tables, based on the input DLL program, to a single file. The configured ADL Tool for the given diagram type, a shellscript, invokes the generic ADL Tool providing the generic ADL Tool with the name of the diagram modeling tables, the name of the specification database, and the name of the output file to write the ICOLA PDI object code to. The configured ADL Tool shellscript has the name of this diagram modeling tables file hard-coded as an input to the the generic ADL Tool. The configured ADL Tool, however, requires the name of the specification database, and the name of the output file for the object code, from the developer each time the developer invokes a configured ADL Tool. When the generic ADL Tool is invoked, it reads in the diagram modeling tables from the file and the developer-named specification database. The generic ADL Tool tracks the relative placement of the icons, and resolves possible placement conflicts, according to the rules in the diagram modeling tables. The generic ADL Tool generates ICOLA PDI code which defines the icons (via the icon library), constrains the icons relative to one another (placement algebra), and connects the required icons (edge routing). The generated code is input to ICOLA which proceeds to layout the diagram. This diagram is laid out for the developer who can then make basic changes to the layout to improve the aesthetics.

CHAPTER 6

Evaluation of the ADL System

6.1 Introduction

The main goals of this thesis included the need to provide an evaluation of:

- the isolated placement and routing strategy,
- the viability of the placement algebraic approach to icon placement, and
- the applicability of the A* search heuristic to edge routing.

This chapter provides some diagrams generated by the prototype and discusses them based on some specific and more general evaluation criteria. The diagrams were evaluated by myself and a small group of analysts within the department of Computer Science. The results of this evaluation provide indications of which aspects of the chosen approach were reasonable and which aspects require additional research.

Section 6.2 describes the evaluation criteria applied to the diagrams. Section 6.3 describes the Data Flow EDL method definition and its corresponding DLL model. This section then presents three specification databases and their corresponding ADL system-generated diagrams. Each diagram is evaluated according to the objective criteria discussed in Section 6.2.

Section 6.4 discusses the Entity Relationship EDL method in the same fashion as the Data Flow method was discussed in Section 6.3.

An overall evaluation of all of the diagrams generated will be presented in Section 6.5, after all of the diagrams have been discussed. The overall evaluation is based on the subjective criteria defined in Section 6.2.

6.2 Evaluation Criteria

The diagram examples for the two methods were evaluated based on objective and subjective readability criteria. The objective evaluation criteria were used to get a rating of the diagrams. The subjective criteria were used to allow the evaluating analysts to comment on aspects of the diagram which were not handled by the objective criteria. The evaluation took the form of a questionnaire in which each analyst was asked three questions. The questionnaire is provided in appendix B.

The first question involved an evaluation of the diagrams based on objective criteria. The analysts were asked to rate each diagram on a scale of 1 to 4 (1 = poor, 2 = fair, 3 = good, 4 = excellent) according to the following objective criteria. The objective evaluation criteria involved eight different aspects. These aspects were:

1. AREA — The AREA criterion deals with the spacing of the diagram. This criterion asks the question: "Does the diagram occupy an appropriate amount of space?"

2. BALANCE — For a diagram to be balanced, it should occupy its space in a reasonable fashion. A typical question here is: "Does the diagram seem lopsided or uneven ?" This aspect also addresses how well icons were placed relative to one another via the placement algebra.

3. BENDS — The ADL system prototype implemented the grid routing standard. This implies that there are right angle bends in the diagram's arcs. The BENDS criteria asks: "Are the bends in the arcs on the diagram appropriate?"

4. DIMENSION — The definition and generation of icons in the ADL system was novel in the sense that the definition of the icons were made via a constraint

language (The ICOLA PDI language). This meant that the icons could be resized, to a certain degree, to fit in large labels. The DIMENSION criterion, therefore, asks the question: "Do icons of the same type differ in size in such a way as to distract from their meaning or importance on the diagram?"

5. LENGTH — The LENGTH criterion asks the question: "Are the lengths of the arcs on the diagram appropriate?"

6. ICON LABELS — The placement of meaningful labels on icons is an important aspect of diagram layout. Thus, the question asked here is: "Are the labels for icons meaningful and placed well?"

7. ARC LABELS — Arcs sometimes also require a meaningful and well-placed label. This aspect asks the question: "Are the labels for arcs meaningful and placed well?"

8. OVERALL — An overall rating of the diagram provides an indication of how well received the diagrams were with respect to the objective criteria listed and other criteria which were not listed but relevant to the evaluating analyst.

Along with the above criteria, the number of arcs not routed is also provided for each diagram. This criterion was not on the questionnaire but was determined by the author for each diagram and presented in the evaluation.

The second question on the questionnaire involved an evaluation of the diagrams based on a more subjective criteria. The subjective criteria involved determining the acceptability of the diagrams within a larger framework. The ADL system prototype was meant to provide a developer with a first draft visualization of a specification database. The resulting diagrams are not meant to be the final product. The ADL system could be used in several ways within the software life cycle. These ways are:

- First Draft Generator — The ADL system could generate first draft diagrams of specification databases. The specification databases would have been created from text input, for example, an SSL program.

- Alternate View Diagram Generator — The ADL system could regenerate diagrams for specification databases which were already drawn out via graphical editor tools, thus providing an alternate view of the same information.

- Transformed Specification Viewer — The Metaview metasystem has a process modeling underlying infrastructure. This means that an entire software development process can be represented. Therefore, specification databases for one phase of the process (corresponding to one method) can be transformed into specification databases corresponding to the another phase in the process. This transformed specification database can then be laid out by the ADL system in order to provide a visualization of the new phase's artifacts.

The third, and final, question also involved an evaluation of the diagrams based on subjective criteria. The subjective criteria involved here is the overall impression of the diagrams. This criterion is a catch-all to allow the evaluating analysts the ability to provide general comments on their perception of the diagrams. For example, their opinion of the strengths and weaknesses of the diagrams.

6.3 Data Flow Diagrams

Data Flow diagrams are used quite extensively in several software development paradigms. This section presents the formal EDL definition of the Data Flow method, the associated DLL model, three SSL-defined Data Flow specification databases and their respective diagrams, and the evaluation of each diagram according to the evaluation criteria defined in Section 6.2.

6.3.1 EDL Definition

The formal EDL definition of the Data Flow method is presented in Figure 6.1. This method definition is similar to the method defined in Section 4.2. The method defines the entity, relationship, and aggregate types, as well as the associated attributes for each type, for the Data Flow Diagram software method. This definition is the current model of the Data Flow method as encoded in Metaview. The formal definition includes entity types which represent the processes, data stores, interfaces, stored and flowing data, along with the relationship types which comprise a Data Flow Diagram method.

6.3.2 A DLL Model of the Data Flow Diagram

This section discusses the DLL-modeled Data Flow Diagram. Figure 6.2 defines the layout rules for the Data Flow Diagram method defined in Figure 6.1. The DLL program defines most of the graphical properties of all of the graphical components of the diagram type being modeled. The DLL code explicitly maps the conceptual entity and relationship types to their icon and edge type graphical counterparts. The placement algebra rules for each edge are then encoded using the placement algebra defined in Section 4.3. In the DLL program of Figure 6.2 a top down orientation was encoded.

6.3.3 Sample Data Flow Diagrams

The following two samples illustrate the functionality of the ADL system. The first sample is a system which represents the basic transactions which take place in a library. The second example is a customer ordering system in which the basic transactions are represented on one Data Flow diagram and a second Data Flow diagram illustrates the aggregation of one of the first diagram's process bubbles.

The samples are given by presenting the specification database, the diagram, and then a discussion of the analysts' evaluation.

6.3.3.1 Sample One: A Library System

The specification database in Figure 6.3 defines an instance of the Data Flow EDL method definition for a library system. The corresponding Data Flow diagram generated by the ADL system is illustrated in Figure 6.4.

The Library Data Flow example is an example of a medium-sized diagram. There

```
ENVIRONMENT_NAME dfd;

  ENTITY_TYPE universal GENERIC
    ATTRIBUTES (description: TEXT);
  ENTITY_TYPE data_object GENERIC IS_A universal,
    data_flow IS_A data_object,
    stored_data IS_A data_object,
    interface IS_A universal;
  ENTITY_TYPE process IS_A universal
    ATTRIBUTES (id_number: STRING,
      form: STRING)
    BECOMES process_explosion
    CONNECTION (sends, changes, reads);
  ENTITY_TYPE data_store IS_A universal
    ATTRIBUTES (id_number: STRING,
      form: STRING);

  VALUE_TYPE times_per_unit:
    RECORD
      quantity: INTEGER;
      unit: (second, minute, hour, day, week, month, year); // enumeration value type
    END RECORD;

  RELATIONSHIP_TYPE sends
    ATTRIBUTES (frequency: times_per_unit)
    ROLES (source, data, destination)
    PARTICIPANTS
      (process, data_flow, process — interface)
      (interface, data_flow, process);
  RELATIONSHIP_TYPE stores
    ROLES (store_name, data:list)
    PARTICIPANTS
      (data_store, stored_data);
  RELATIONSHIP_TYPE access GENERIC
    ROLES (p_name, data, store)
    PARTICIPANTS
      (process, stored_data, data_store);
  RELATIONSHIP_TYPE changes IS_A access;
  RELATIONSHIP_TYPE reads IS_A access;

  AGGREGATE_TYPE any_level GENERIC
    ATTRIBUTES (description: TEXT)
    COMPONENTS (process, data_object, data_store, interface,
      sends, stores, access);
  AGGREGATE_TYPE context_diagram IS_A any_level;
  AGGREGATE_TYPE process_explosion IS_A any_level;

  RELATIONSHIP_TYPE derived_from
    ROLES (derived_add, source_agg)
    PARTICIPANTS
      (process_explosion, process_explosion);

  RELATIONSHIP_TYPE has_subparts
    ROLES (superpart, subpart)
    PARTICIPANTS
      (data_flow, data_flow)
      (stored_data, stored_data)
      (interface, interface)
      (data_store, data_store);
```

Figure 6.1: EDL Definition of Data Flow Method

```
PROGRAM: dfd;
DIAGRAM_TYPE_NAME: process_explosion;
ORIENTATION: TOP_DOWN;
STANDARD: GRID;

ICON_TYPES: {LABELED (datastore_icon, interface_icon, process_icon)};
ARC_TYPES: {LABELED UNI_DIRECTIONAL (uni_flow_sends, uni_flow_reads,
            uni_flow_stores),
        LABELED BI_DIRECTIONAL (bi_flow_changes)};

MAP_REL
    uni_flow_sends TO sends
    SENDER IS ROLE 1 RECEIVER IS ROLE 3;
MAP_REL
    uni_flow_reads TO reads
    SENDER IS ROLE 1 RECEIVER IS ROLE 3;
MAP_REL
    uni_flow_stores TO stores
    SENDER IS ROLE 1 RECEIVER IS ROLE 3;
MAP_REL
    bi_flow_changes TO changes
    SENDER IS ROLE 1 RECEIVER IS ROLE 3;
MAP_ENTITY datastore_icon TO data_store;
MAP_ENTITY process_icon TO process;
MAP_ENTITY interface_icon TO interface;

RULE_BLOCK

    ASSOCIATE int: interface_icon;
    ASSOCIATE pro: process_icon;
    ASSOCIATE ds: datastore_icon;

    RELATIONSHIP uni_flow_sends(_,_)
        CASE int,pro:
            LEFT_OF(int,pro) AND ABOVE(int,pro);
        ENDCASE
        CASE pro,int:
            RIGHT_OF(pro,int) AND BELOW(pro,int);
        ENDCASE
        CASE pro,pro:
            RIGHT_OF(pro,pro);
        ENDCASE
    ENDRELATIONSHIP

    RELATIONSHIP uni_flow_reads(_,_)
        CASE pro, ds:
            LEFT_OF(pro, ds) AND ABOVE(pro, ds);
        ENDCASE
        CASE int,pro:
            LEFT_OF(int,pro) AND ABOVE(int,pro);
        ENDCASE
        CASE pro,int:
            RIGHT_OF(pro,int) AND BELOW(pro,int);
        ENDCASE
        CASE pro,pro:
            RIGHT_OF(pro,pro) AND BELOW(pro,pro);
        ENDCASE
    ENDRELATIONSHIP

    RELATIONSHIP bi_flow_changes(_,_)
        CASE pro, ds:
            LEFT_OF (pro, ds);
        CASE ds, pro
            RIGHT_OF (ds, pro);
        ENDCASE
    ENDRELATIONSHIP

    CONFLICT:
    {
        uni_flow_sends(pro, pro), uni_flow_sends(pro, pro) ;
    }
    RESOLUTION_MAPPING:
    {
        uni_flow_sends(SENDER) = uni_flow_sends(RECEIVER):
            PRIORITY_IS uni_flow_sends(SENDER);
    }

ENDBLOCK

END
```

Figure 6.2: DLL Model of Data Flow Diagram

```
BEGIN process_explosion library_system
    library_patron interface;
    librarian interface;
    database_access process;
    query_access process;
    library_holdings data_store;
    sends(library_patron, query, query_access);
    sends(query_access, query_reply, library_patron);
    sends(librarian, update_request, update_access);
    sends(update_access, parsed_request, database_access);
    sends(query_access, parsed_query, database_access);
    sends(database_access, reply_info, query_access);
    reads(library_holdings, holdings_info, database_access);
    reads(database_access, holdings_info, library_holdings);
END library_system
```

Figure 6.3: Library System Specification Database

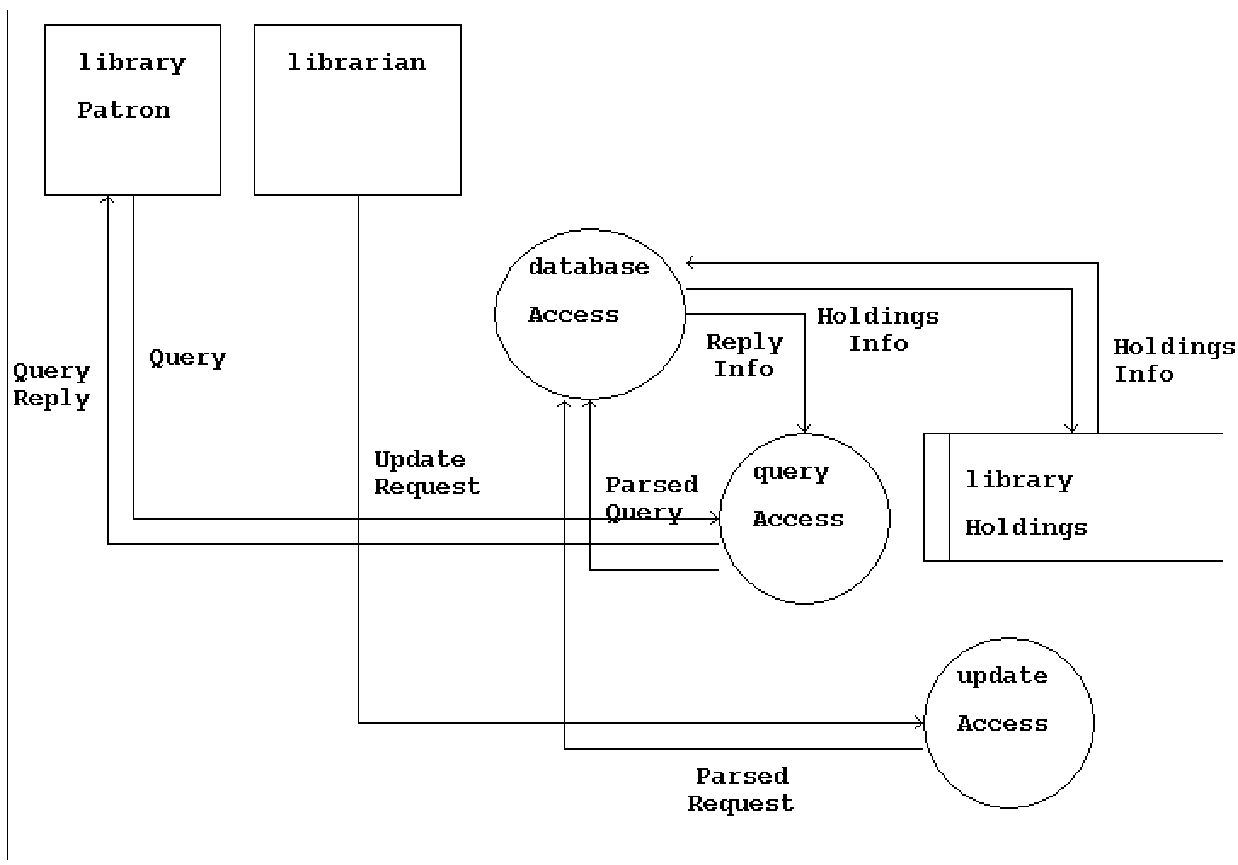


Figure 6.4: Library System Data Flow Diagram

Table 6.1: Analysts' Evaluation of the Library System DFD

Criterion	Analyst 1	Analyst 2	Analyst 3	Analyst 4	Analyst 5	Analyst 6
AREA	3	4	3	4	4	3
BALANCE	2	4	3	3.5	4	2
BENDS	4	4	2	3	4	2
DIMENSION	4	4	4	4	3	4
LENGTH	2	4	3	1.5	3	1
ICON LABELS	4	4	3	3.5	3	4
ARC LABELS	2	4	3	2	3	2
OVERALL	3	4	3	2.5	3	2

were no unrouted arcs on this diagram. The analysts' objective evaluation results for this diagram are shown in Table 6.1. This table reflects the analysts' results for the first question of the questionnaire. The majority of the analysts' ranked this diagram above average for all of the criteria. This, however, does not mean that the diagram is flawless. The diagram faired the best with respect to the AREA, DIMENSION and ICON LABELS criteria. This would imply that the placement of the icons does not hinder the readability but enhances it. The ARC LABELS criterion faired the worst. This means that the arc labelling detracted from the readability of the diagram. In some instances the labels were ambiguous in the sense that it was unclear as to which arc a label was attached. This is a result of the isolated placement and routing approach. The criteria of BALANCE, BENDS, and LENGTH faired about average. An average rating means that these aspects of the diagram were not detracting from the readability of the diagram.

The OVERALL rating of the diagram was higher than average; therefore, the diagram was able to convey its intended meaning to the analysts.

6.3.3.2 Sample Two: Customer Ordering System

The specification database in Figure 6.5 defines an instance of the Data Flow EDL method definition for a customer ordering system. The corresponding Data Flow diagram generated by the ADL system is illustrated in Figure 6.6.

The Customer Ordering System Data Flow example is an example of a large-sized diagram. There were three unrouted arcs on this diagram. The analysts' objective

```
BEGIN process_explosion process_customer_order;
    customers interface;
    accounts interface;
    issue_order_acknowledge process;
    check_available_stock process;
    accept_order process;
    makeup_sales_record process;
    match_stock_with_order process;
    maintain_stock_inventory process;
    accept_customer_payment process;
    price_list data_store;
    transaction_record data_store;
    custorder_record data_store;
    sales_record data_store;
    customer_record data_store;
    sends(match_stock_with_order, newstock_arrnotice, customers);
    sends(accept_customer_payment, payment_details, accounts);
    sends(accounts, payment_receipt, accept_customer_payment);
    sends(maintain_stock_inventory, newstock_details, match_stock_with_order);
    sends(check_available_stock, approved_custorder, makeup_sales_record);
    sends(issue_order_acknowledge, order_acknowledge, customers);
    sends(customers, payment_details, accept_customer_payment);
    sends(accept_customer_payment, invoice, customers);
    sends(check_available_stock, rejected_order, customers);
    sends(accept_order, customer_order, check_available_stock);
    sends(customers, customer_order, accept_order);
    sends(customers, customer_info, accept_order);
    sends(check_available_stock, stock_request, maintain_stock_inventory);
    sends(maintain_stock_inventory, stock_details, check_available_stock);
    sends(maintain_stock_inventory, rejected_stkrequest, check_available_stock);
    sends(maintain_stock_inventory, stkorder_requisition, check_available_stock);
    sends(check_available_stock, stkorder_requisition, issue_order_acknowledge);
    reads(match_stock_with_order, order_record, custorder_record);
    reads(match_stock_with_order, customer_details, customer_record);
    reads(makeup_sales_record, customer_details, customer_record);
    reads(makeup_sales_record, item_price, price_list);
    reads(accept_customer_payment, unpaid_sales_rec, sales_record);
    changes(accept_order, customer_details, customer_record);
    changes(check_available_stock, order_record, custorder_record);
    changes(makeup_sales_record, unpaid_salesrec, sales_record);
    changes(accept_customer_payment, transaction_details, transaction_record);
END process_customer_order;
```

Figure 6.5: Process Customer Order System Data Flow Specification Database

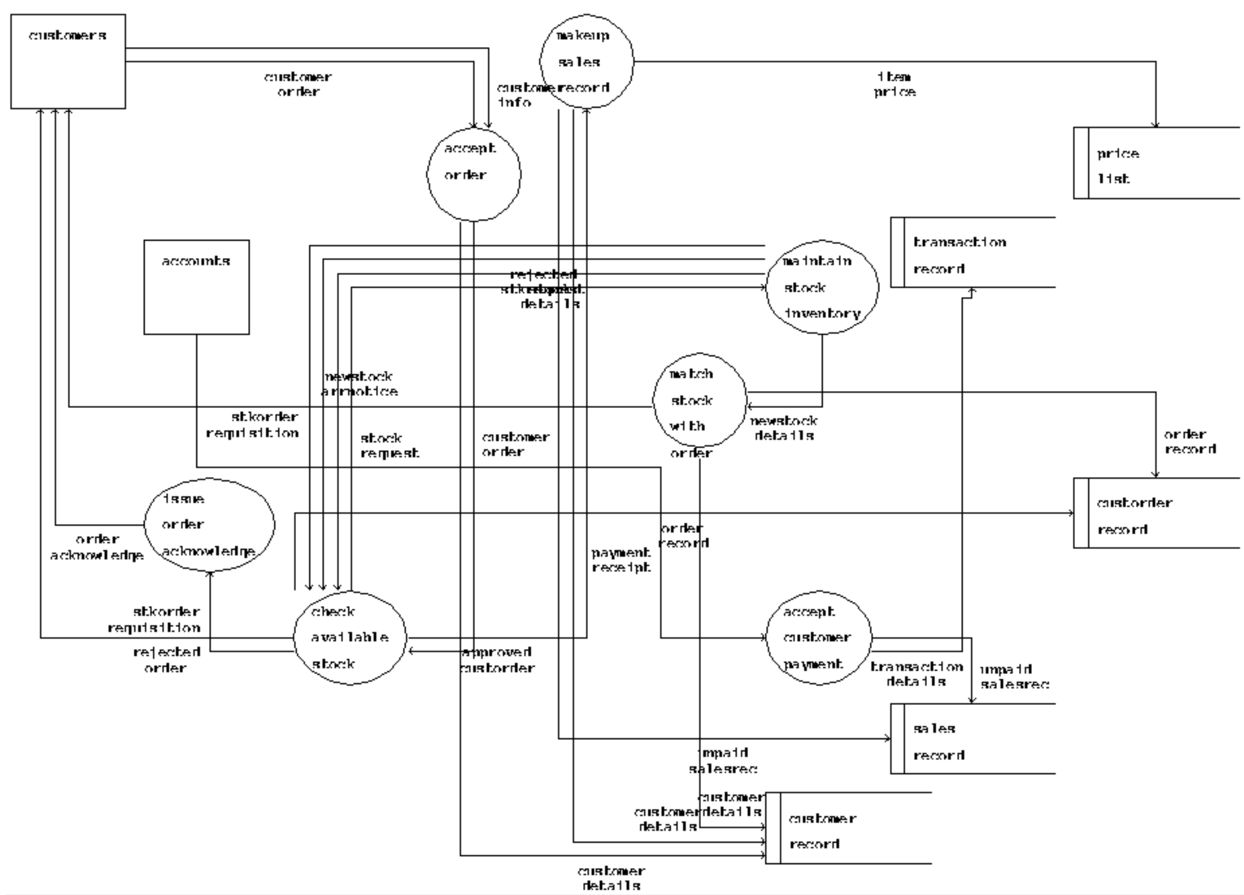


Figure 6.6: Process Customer Order Data Flow Diagram

Table 6.2: Analysts' Evaluation of the Customer Ordering System DFD

Criterion	Analyst 1	Analyst 2	Analyst 3	Analyst 4	Analyst 5	Analyst 6
AREA	3	4	3	4	4	4
BALANCE	3	4	3	4	4	3
BENDS	2	4	2	3	2	1
DIMENSION	2	4	4	4	2	4
LENGTH	2	4	2	1.5	4	1
ICON LABELS	4	3	3	4	3	4
ARC LABELS	1	2	1	2	1	1
OVERALL	1	3.75	1	2	2	2

evaluation results for this diagram are shown in Table 6.2. The majority of the analysts' ranked this diagram average for most of the criteria. The diagram faired the best with respect to the AREA, BALANCE, DIMENSION and ICON LABELS criteria. This would imply that the placement of the icons does not hinder the readability but enhances it. The ARC LABELS criterion faired the worst. This means that the arc labelling detracted from the readability of the diagram. In many instances the labels were ambiguous in the sense that it was unclear as to which arc a label was attached. This, again, is a result of the isolated placement and routing approach. The criteria of BENDS, and LENGTH faired about average or a little lower in some instances. This rating implies that the routing of the arcs detracted from the readability of the diagram.

The OVERALL rating of the diagram was lower than average; therefore, the diagram was not able to convey its intended meaning to the analysts primarily due to the problems with respect to the arc routing and labelling.

The specification database for the aggregation of the makeup_sales_record process appears in Figure 6.7 and the corresponding Data Flow diagram generated by the ADL system is illustrated in Figure 6.8.

The Makeup Sales Record Data Flow example is another example of a medium sized diagram. There were no unrouted arcs on this diagram. The analysts' objective evaluation results for this diagram are shown in Table 6.3. The majority of the analysts' ranked this diagram above average for all of the criteria. This would imply that the placement of the icons and arcs does not hinder the readability but enhances it. The BENDS criterion faired the worst. This means that, in this case, the routing

```
BEGIN process_explosion makeup_sales_record
    price_list data_store;
    customer_record data_store;
    sales_record data_store;
    check_available_stock process;
    prepare_sales_record process;
    price_sales_record process;
    total_sales_record process;
    record_customer_info process;
    sends(prepare_sales_record, sales_draft, price_sales_record);
    sends(check_available_stock, approved_order, prepare_sales_record);
    sends(price_sales_record, , priced_record, total_sales_record);
    sends(total_sales_record, totalled_record, record_customer_info);
    sends(price_sales_record, item_price, price_list);
    sends(record_customer_info, customer_details, customer_record);
    sends(record_customer_info, unpaid_salesrec, sales_record);
END makeup_sales_record
```

Figure 6.7: makeup_sales_record Process Specification Database

of the arcs had some bends which did not detract from the readability but were unnecessary.

The OVERALL rating of the diagram was higher than average; therefore, the diagram was able to convey its intended meaning to the analysts.

Table 6.3: Analysts' Evaluation of the Makeup Sales Record DFD

Criterion	Analyst 1	Analyst 2	Analyst 3	Analyst 4	Analyst 5	Analyst 6
AREA	3	4	3	4	4	4
BALANCE	4	4	3	3.5	4	3
BENDS	3	4	2	3	3	3
DIMENSION	4	4	4	4	3	4
LENGTH	4	4	4	3	4	3
ICON LABELS	4	4	3	4	4	4
ARC LABELS	4	4	3	4	4	3
OVERALL	4	4	3	3	4	3

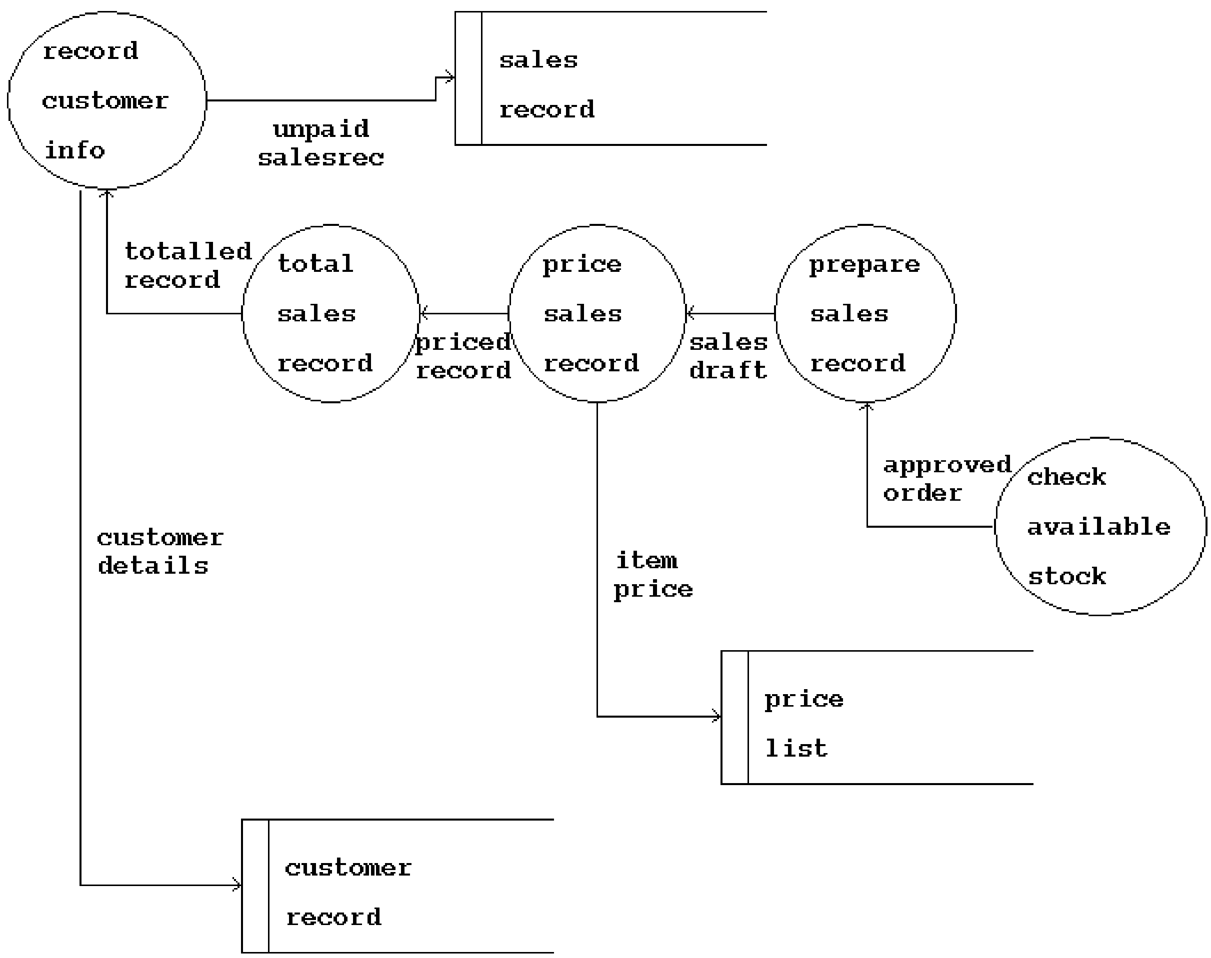


Figure 6.8: Makeup Sales Record Data Flow Diagram

6.4 Entity Relationship Diagrams

Entity Relationship diagrams are used quite extensively in the software development process. They are used as a tool to visualize the inter-relationships among data. This section presents the formal EDL definition of the Entity Relationship method, the associated DLL model, two SSL-defined Entity Relationship specification databases and their respective diagrams, and the evaluation of each diagram according to the evaluation criteria previously defined.

6.4.1 EDL Definition

The formal EDL definition of the Entity Relationship method is presented in Figure 6.9. The method defines the entity, relationship, and aggregate types, as well as the associated atttributes for each type, for the Entity Relationship Diagram software method. This definition is the current model of the Entity Relationship method as encoded in Metaview. The formal definition includes entity types which represent the entity and relationship objects along with the relationship types which comprise an Entity Relationship Diagram method.

6.4.2 A DLL Model of the Entity Relationship Diagram

This section discusses the DLL-modeled Entity Relationship Diagram. Figure 6.10 defines the layout rules for the Entity Relationship Diagram method defined in Figure 6.9. The DLL program defines most of the graphical properties of all of the graphical components of the diagram type being modeled. The DLL code explicitly maps the conceptual entity and relationship types to their icon and edge type graphical counterparts. The placement algebra rules for each edge are then encoded using the placement algebra defined in Section 4.3.

```
ENVIRONMENT_NAME wmrt;
  AGGREGATE_TYPE data_schema IS_A universal_agg
  COMPONENTS (object_type, relationship, dummy, association, is_supertype, binds);

  ENTITY_TYPE object_type IS_A universal_entity
    ATTRIBUTES (meaning: text, composition: text, objecttype: text);
  ENTITY_TYPE relationship IS_A universal_entity
    ATTRIBUTES (relationship_name, string);
  ENTITY_TYPE dummy;

  RELATIONSHIP_TYPE binds IS_A universal_rel
    ROLES (relationship, entity, dummy)
    PARTICIPANTS
      (relationship, object_type, dummy);
  RELATIONSHIP_TYPE is_supertype IS_A universal_rel
    ROLES (supertype, the_rest)
    PARTICIPANTS
      (object_type, relationship);
  RELATIONSHIP_TYPE association IS_A universal_rel
  ROLES (object, relationship)
  PARTICIPANTS
    (object_type, relationship);
```

Figure 6.9: EDL Definition of Entity Relationship Method

6.4.3 Sample Entity Relationship Diagrams

The following two samples illustrate the functionality of the ADL system. The first sample is a system which represents the classification of a book in a library. The second example is a representation of an airport.

The samples are given by presenting the specification database, the diagram, and then a discussion of the analysts' evaluation.

6.4.3.1 Sample One: A Book Classification

The following specification database defines an instance of the Entity Relationship EDL method definition for book classification. The corresponding Entity Relationship diagram generated by the ADL system is illustrated in Figure 6.11.

```
BEGIN data_schema book_classification;

  object_type book;

  object_type fiction;

  object_type non_fiction;

  relationship relationship0
```

```
PROGRAM: wmrt;
DIAGRAM_TYPE_NAME: wmrt;
ORIENTATION: TOP_DOWN;
STANDARD: GRID;

ICON_TYPES: {LABELED (entity_icon, relationship_icon)};
ARC_TYPES: {NON_LABELED NON_DIRECTED (association_uni_flow,
        is_supertype_uni_flow, binds_uni_flow)};

MAP_REL
    association_uni_flow TO association
    SENDER IS ROLE 1 RECEIVER IS ROLE 2;
MAP_REL
    is_supertype_uni_flow TO is_supertype
    SENDER IS ROLE 1 RECEIVER IS ROLE 2;
MAP_REL
    binds_uni_flow TO binds
    SENDER IS ROLE 1 RECEIVER IS ROLE 2;
MAP_ENTITY entity_icon TO object_type;
MAP_ENTITY
    relationship_icon TO relationship
    LABEL IS  relationship_name ;

RULE_BLOCK

    ASSOCIATE ent: entity_icon;
    ASSOCIATE rel: relationship_icon;

    RELATIONSHIP association_uni_flow(_,_)
        CASE ent, rel:
            BELOW(ent, rel);
        ENDCASE
    ENDRELATIONSHIP

    RELATIONSHIP is_supertype_uni_flow(_,_)
        CASE ent, rel:
            ABOVE(ent, rel);
        ENDCASE
    ENDRELATIONSHIP

    RELATIONSHIP binds_uni_flow(_,_)
        CASE ent,rel:
            LEFT_OF(ent,rel);
        ENDCASE
        CASE rel,ent:
            RIGHT_OF(rel,ent) AND ABOVE(rel,ent);
        ENDCASE
    ENDRELATIONSHIP

ENDBLOCK

END
```

Figure 6.10: DLL Model of Entity Relationship Diagram

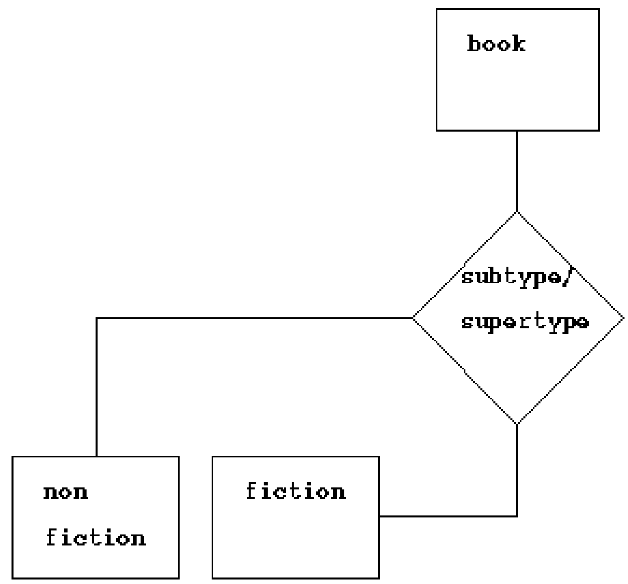


Figure 6.11: Book Entity-Relationship Diagram

Table 6.4: Analysts' Evaluation of the Book Classification ERD

Criterion	Analyst 1	Analyst 2	Analyst 3	Analyst 4	Analyst 5	Analyst 6
AREA	4	4	3	4	4	3
BALANCE	4	4	3	2	3	2
BENDS	4	4	1	4	2	1
DIMENSION	4	4	4	4	4	4
LENGTH	4	4	4	3	3	1
ICON LABELS	3	4	3	4	3	4
ARC LABELS	-	-	-	-	-	-
OVERALL	4	4	3	2.5	3	2

```
    (relationship_name = "subtype/supertype");
  binds(relationship0, fiction);
  binds(relationship0, non_fiction);
  is_supertype(book, relationship0);
END book_classification
```

The book classification Entity Relationship diagram is an example of a small diagram. There were no unrouted arcs on this diagram. The analysts' objective evaluation results for this diagram are shown in Table 6.4. The majority of the analysts' ranked this diagram above average for all of the criteria. This would imply that the placement of the icons and arcs does not hinder the readability but enhances it. The ARC LABEL criterion for this example and the next were not applicable because Entity Relationship diagrams don't have labels on their arcs. The BENDS criterion faired the worst. This means that, in this case, the routing of the arcs had some anomolous bends which did not detract from the readability but were unnecessary.

The OVERALL rating of the diagram was higher than average; therefore, the diagram was able to convey its intended meaning to the analysts.

6.4.3.2 Sample Two: An Airport Model

The following specification database defines an instance of the Entity Relationship EDL method definition for an airport. The corresponding Entity Relationship diagram generated by the ADL system is illustrated in Figure 6.12.

```
BEGIN data_schema airport
  object_type pilot;
  object_type aircraft;
  object_type runway;
  relationship relationship1
    (relationship_name “intersect”);
  relationship relationship2
    (relationship_name “fly”);
  relationship relationship3
    (relationship_name “land”);
  relationship relationship4
    (relationship_name “take_off”);
  binds(relationship1,runway)
  binds(relationship1,runway)
  binds(relationship3,aircraft)
```

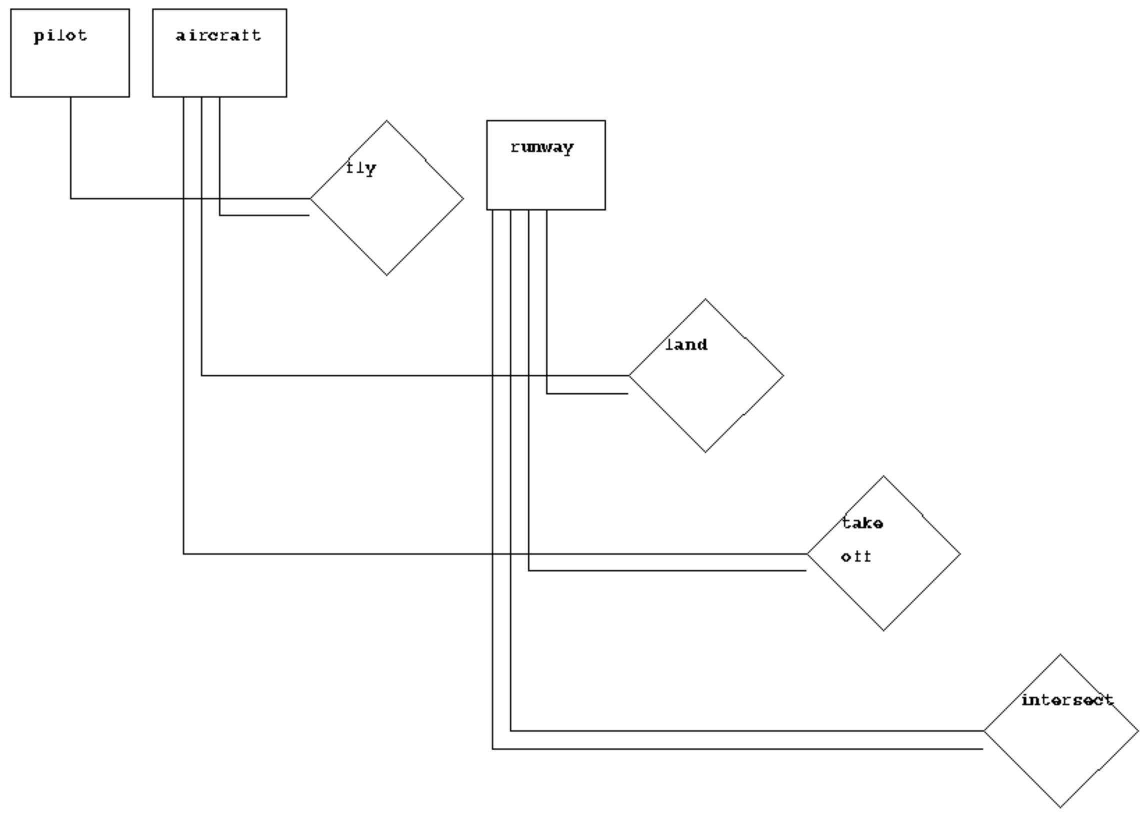


Figure 6.12: Airport Entity-Relationship Diagram

binds(relationship2,pilot)

binds(relationship2,aircraft)

binds(relationship3,runway)

binds(relationship4,aircraft)

binds(relationship4,runway)

END airport

The airport Entity Relationship diagram is an example of a medium-sized diagram. There were no unrouted arcs on this diagram. The analysts' objective evaluation results for this diagram are shown in Table 6.5. The majority of the analysts' ranked this diagram average for all of the criteria. The AREA, DIMENSION, ICON LABELS were rated higher than average by most of the analysts. The BALANCE criterion, however, was ranked low. Therefore, the placement of the icons detracted from the

Table 6.5: Analysts' Evaluation of the Airport ERD

Criterion	Analyst 1	Analyst 2	Analyst 3	Analyst 4	Analyst 5	Analyst 6
AREA	2	4	3	3	4	3
BALANCE	2	3	1	2.5	1	2
BENDS	2	3	1	3.5	2	1
DIMENSION	4	4	4	4	4	4
LENGTH	3	3	2	1.5	1	1
ICON LABELS	3	3	3	3	2	3
ARC LABELS	-	-	-	-	-	-
OVERALL	3	3	2	2	1	2

readability of the diagram because of the somewhat lopsided nature of the layout. This was a result of the placement algebra and the fact that the current implementation has the placement algebra being performed on a per relationship basis. Therefore, when there are very few relationship types, as is the case with Entity Relationship diagrams, then the placement algebra approach will produce somewhat unbalanced layouts. The BENDS and LENGTH criteria faired low as well. This means that, in this case, the routing of the arcs had some anomolous bends which also detracted from the readability of the diagram. This was a result of both the isolated placement and routing as well as the A* routing algorithm.

The OVERALL rating of the diagram was lower than average; therefore, the diagram was not able to convey all of its intended meaning to the analysts.

6.5 Overall Subjective Evaluation

In this section, all of the diagrams will be evaluated according to the subjective criteria of Section 6.2. The last two questions of the questionnaire dealt with these criteria.

Considering that these criteria are so subjective, the responses received from the analysts were quite diverse. Some felt that the diagrams were quite good others had the opionion that some of them needed more work than they would be willing to do with a graphical editor. With this in mind, the analysts comments and suggestions will be discussed.

The second question of the questionnaire dealt with the adequacy of the diagrams as a first draft for the analysts to make improvements. Three of the analysts felt that all of the diagrams were quite reasonable as first draft versions of the corresponding specification databases. One analyst stated that he felt that the modifications necessary were quite minor. Another analyst felt that the diagram qualities were good but they needed some improvement to be transformed into a final version. He felt that this type of first draft tool would reduce development labour. Another analyst would accept all diagrams as a first draft provided that a reasonable graphic editor could be invoked to make the changes so that things like arc labels would stay with their arcs, and arcs should not lose connections to their respective icons when either the arc or the icon is moved. The developer should also be allowed to reposition labels relative to the arc or icon to which they're attached.

Three of the analysts had difficulty in accepting diagrams two and five as a first draft for various reasons. One reason for diagram two being rejected was the poor labelling of the arcs and too many crossovers. Diagram two was the largest diagram generated by the ADL system and revealed some good and bad aspects of the prototype system. The balance of the diagram turned out quite well; however, this was countered with a poor labelling of the arcs and many distracting crossovers. The generation of this diagram showed that the A* edge routing algorithm's routing criteria should have included a crossover minimization criterion. A decision was made early in the design of the prototype to prioritize the minimization of bends as oppose to minimizing crossovers because of the fact that the two generally work in opposition to one another in many instances. It was felt that this would result in more readable diagrams. Diagram two, as well as other diagrams, show that effort needs to be put into the minimization of crossovers as well as the minimization of bends. This, however, should not be taken lightly because Gansner [11] states that the minimization of edge crossings in a layout is computationally intractable. Therefore in order to attempt to minimize edge crossings, one must be satisfied with only using a heuristic solution to the problem. This is not surprising because when one deals with the subjectivity of diagram layout one must employ heuristics at almost every turn because there currently is no guaranteed computational means of attaining optimal layouts.

Diagram five illustrates a problem inherent in the placement algebra. This diagram has only a couple of relationship and entity types. With such a small number of types to be represented on a diagram, the placement algebra modeled in the DLL program has limited options with which to place icon types relative to one another. This is because of the fact that the DLL program places icons relative to one another on a per relationship basis. Therefore, the fewer the choices, the more unbalanced a diagram will most likely be in the current implementation of the ADL system prototype.

The third question asked the analysts to give their overall impression of the diagrams generated by the ADL system. The overall impression of the diagrams was good. The analysts did tend to pick out a few common strengths. These common strengths for the diagrams include the balance of the icons for most of the diagrams, the uniformity of the icons, and the overall clarity. Some analysts felt that the arcs were reasonably placed on the diagram and connected to their respective icons. Other analysts especially liked the icons and their size relative to their labels. Yet other analysts felt that the ability to directly generate diagrams from formal specifications was a great advantage to reduce overhead and human error in manually generating them. The ability to generate diagrams from formal specifications was felt to be a major advantage. In the case of a redesign of a system, the ADL system could be used to regenerate the diagrams from the formal design specifications to save the developers the time to redraw them manually.

One analyst expecially liked the layout of diagram two. He felt that the organization was good in general and that there was a good use of page layout area.

The analysts also found some weaknesses in the diagrams. Some of these weaknesses were alluded to earlier in the discussion of question one but will be discussed here for completeness. The major weaknesses discovered by this evaluation were the poor labelling of the arcs and the large numbers of unnecessary edge crossings on a couple of the diagrams.

A couple of the analysts noted that there were a few unnecessary bends in certain arcs. These unnecessary bends were not considered major weaknesses by the analysts but they were flagged as anomolous. These unnecessary bends were the result of the A* routing algorithms pre-determined endpoint property. In some situations the search finds a path but for reasons inherent to the algorithm this path has a little bend near the end close to an icon.

One analyst felt that the placement of the icons was not optimal. Optimal placement of icons, however, was not acheivable via the placement algebra approach due to its heuristic nature.

Other analysts noted another weakness to be the unconnectedness of some of the arcs to their respective icons. This was due to a limitation in the implementation. Although some of the icons appeared to have a non-rectangular shape to them, the reality was that all icons were enclosed by a bounding rectangualr box which the routing algorithm saw as an obstruction and therefore did not route into it to get at the possibly non-rectangular icon.

Another weakness pointed out was the difficulty to read some of the labels due to the font and point size. This weakness was due to the author's efforts to fit some of the diagrams onto one page for the purpose of readability. The ADL system prototype actually generated all the diagrams with the same font and point size but some diagrams were scaled down and this scaling introduced the font size problems referred to here.

One analyst felt that he would be unwilling to learn a new language in order to generate these diagrams. This is a reasonable concern due to the design nature of the ADL system. The ADL system uses the Diagram Layout Language at the environment level in order to model the diagram type. If the developer was fulfilling both the methods engineer and developer roles then he/she would definately have to learn DLL before using the tool. Learning a new language is never easy but in the case of the ADL system, there is no other obvious way of encoding a model of a diagram type.

CHAPTER 7

Conclusions and Future Work

This chapter begins by presenting a summary of the thesis. The summary is followed by a discussion of the conclusions drawn based on the research performed. The chapter concludes with suggestions for future work and research for the ADL system. The future work will deal with the shortcomings of the implemented system and expand on the design to bring us closer to achieving a generic diagram layout tool for software development.

7.1 Summary of the Thesis

Chapter one introduced the motivation for finding a solution to the automatic diagram layout problem. It presented a high-level description of the Metaview system, and a tool to be built within the Metaview framework for laying out diagrams automatically. Chapter one also defined the goals of the thesis research.

Chapter two examined the Metaview system in some detail. A discussion of the three levels of specification as well as the underlying meta-level data model, the EARA model, was provided.

Chapter three presented some related work in the area of automatic diagram layout. MONDRIAN, GIOTTO, and DAG were presented and discussed. Lee's algorithm was also presented.

Chapter four presented the design of the prototype tool developed as part of the thesis research. The ADL tool's main components and their inter-relationships were presented. The placement algebra approach and its implementation within the Diagram Layout Language was discussed in detail. The ADL Tool and the A* edge router were also presented.

Chapter five discussed the low level design and implementation of the DLL compiler, the ADL tool, and the A* edge router.

Chapter six presented and evaluated five diagrams based on two diagram types. The formal EDL defintion and DLL models were presented for the two diagram types along with corresponding specification databases and diagrams generated from these specification databases via the ADL system. Each diagram was evaluated according to objective and subjective criteria.

7.2 Conclusions

The main goals of the thesis were:

1. Develop a research prototype of an automatic diagram layout tool in which the methods engineer has control over the diagram type to be layed out.

2. Evaluate the research protoype to determine if the approach taken was reasonable.

A research prototype of the ADL system was designed and implemented. These aspects were explained in Chapters 4 and 5, respectively.

The second goal was acheived as well. The evaluation of the diagrams generated by the ADL system was discussed in Chapter 6. Based on that evaluation, the following conclusions can be made about the approach taken for the development of the research prototype discussed in this thesis.

The placement algebra proved to be a good heuristic solution for the placement of icons problem. The evaluation revealed that the overall balance of most diagrams as well as the area occupied seemed to appeal to the evaluating analysts. The one drawback of the placement platform ICOLA was its inability to allow the ADL system to take full advantage of all of the placement algebra operators. This restricted the modeling of a diagram type to make use of only a minimal operator set — *left_of, right_of, above,* and *below*. This restriction, however, did not prevent the generation of some reasonable diagrams. The placement algebra heursitic can lead to some unbalanced diagrams if the Metaview method being modeled is very small. For these diagram types, a developer may still be accepting of the diagrams generated because even a poorly laid out diagram reveals information that is not obvious in a textual specification database. Therefore, the placement algebra concept showed itself to be a useful means of obtaining a visualization of a textual specification.

The A* edge routing heuristic was shown to be a reasonable edge routing algorithm for grid routing. The open problems with this routing method are the distracting edge crossovers, some anomolous bends in the edges routed, and the time it took to find suitable paths for edges.

The decoupling of the placement of icons from the routing of edges proved to be more of a detriment than a benefit in this research because of the fact that some of the more distracting bad points of the generated diagrams could have been circumvented. For example, arc labelling could have been dealt with better if the placement and routing were more tightly coupled because the arc labeling could have been performed during the routing phase where it should have been performed instead of having labels being placed at default locations during the placement phase.

Although there were problems with the diagrams generated by the ADL system prototype, the placement algebra outlined in this thesis, coupled with the A* routing strategy, provided the flexibility, in the author's opinion, for a good starting point for further investigation into the general diagram layout problem.

7.3 Suggestions for Future Work and Research

The ADL system is intended as a generic diagram layout tool. The prototype discussed in this chapter has some limitations which restrict the scope of the types of diagrams that ADL can handle.

The first limitation involves the Diagram Layout Language *grouping* construct. This language construct was designed but not implemented. This means that any diagrams which require some form of *grouping* cannot be represented by the ADL system. Some currently non-representable diagram types include Structure Charts, Decision Trees, and any other type of diagram that requires a hierarchical tiering of icons. The grouping construct should be implemented in future versions of the prototype. This construct could be handled by generating a group table similar to the icon placement table discussed in the implementation. Essentially, one would have a group defined in a DLL program and then the DLL compiler would generate the group table with each group name and a list of icon types which are participants. The DLL program could then make reference to a particular group via its methods engineer-defined name within the placement algebra component of the program. The ADL Tool would then look up the group name within the table to resolve which icons the placement algebra should be referring.

Another limitation involves the DLL *routing standard* construct in which the methods engineer encodes the desired type of edge routing standard to be used in the modeled diagram. Currently, only grid routing is available. The reason for only allowing grid routing as the routing standard was simply the complexity factor. Implementing a generic placement **and** routing strategy proved to be too time consuming, therefore a decision was made to restrict the routing in order to allow more genericity for the placement. This results in a further reduction of scope of the types of diagrams representable by ADL. For example, only the Gane and Sarson class of Data Flow Diagrams, which employ a grid routing standard, can be handled by ADL. Other classes of Data Flow Diagrams which require the curved, or mixed routing standards simply can't be handled by the current ADL prototype. A future version of the ADL system would be able to handle more routing standards.

The ADL system protoype rests on the platform of ICOLA. ICOLA, thus, limited the ADL system with respect to the type of placement algebra operators that could be used within a DLL program. These operators were restricted to *left_of, right_of, above,* and *below*. This turned out not to be a very restrictive limitation because most of the other operators tended to be restricted combinations of these four operators. However, in order to implement the placement algebra in full, the placement constraint resolver needs to be able to handle a larger portion of the placement algebra than just the four operators already mentioned. ICOLA also did not prevent icons from being placed on top of one another, therefore if another constraint was incorporated within ICOLA which prevents this from happening then the ADL Tool could communicate this to ICOLA to cause it to search for overlapping icons and then constrain their placement to be non-overlapping. The direction to which they would be constrained should be encoded in the DLL program.

The ADL System prototype is restricted to the placement of binary edges on a diagram. The Metaview system allows the conceptual representation of m-ary relationships. Future versions of the ADL System should allow the definition of m-ary edges to be placed on a diagram, including edges starting and stopping on the same icon. To accomplish this, the Diagram Layout Language would have to be modified to allow the methods engineer to encode the type of edge, and the roles that the icon types would be fulfilling. The DLL compiler would then generate edge tables to reflect that a certain edge type has a self-directed, binary, or m-ary edge as well as the icon types which will be fulfilling roles in the placed edge. This modification would also require a more intelligent routing algorithm than just the A* search heuristic. The A* algorithm could still be used as a sub-module of the overall router. The overall router should handle the grid standard (via the A* search algorithm)as well as the curved, diagonal, and mixed standards.

The evaluation revealed that the isolation of placement and routing was a detriment. This can be solved by placing constraint solving within the ADL Tool directly. This would result in a tighter coupling of the placement and routing. A practical means of acheiving this could be to pull out the constraint solving engine from ICOLA and dropping it into the ADL Tool. Therefore, the ADL system would then be composed of only two parts instead of the current three. These two parts could be considered the front end (the DLL compiler) and the back end (the ADL Tool). With this new partitioning of the ADL system, further enhancements could be made to the front end's Diagram Layout Language so that more information about the diagram type being modeled can be represented, for example, increase the number of aesthetic constraints, semantic constraints, representative icon types and edge types within the language. Along with the new constraints, the Diagram Layout Language should be able to allow the developer to specifiy how indirect conflicts should be resolved. This could be handled by providing DLL syntax similar to the current direct conflict resolution code, and invoking the coupled constraint solver to determine if there any indirect placement conflicts involving the edge types and icon types specified in the conflict resolution tables generated by the DLL compiler.

The current ADL system does not deal effectively with the problem of aggregation within the Metaview system. This problem occurs when an entity *explodes* into another diagram and certain edges are incident into the aggregate diagram. Essentially, certain relationships appear from the boundaries of the diagram and do not originate from an icon. The ADL system can't handle this problem. Future work needs to be put into resolving this problem so that aggregation can be correctly represented on an aggregated diagram.

The A* edge router had time efficiency problems, therefore, an investigation of alternative grid routing algorithms needs to be performed. One such algorithm is the iterative-deepening search heuristic which has been applied to game tree searches quite successfully. This algorithm was first applied to chess playing in 1977 by Slate and Atkin (CHESS 4.5) and it is discussed in Rich and Knight [22]. The iterative-deepening algorithm couples the best aspects of breadth-first search and the depth-first search. The algorithm finds shallow solutions in time commensurate with a breadth-first search while keeping the space used commensurate with a depth-first search.

The ADL system was intended as a first draft diagram generating tool. Therefore, developers will require a feedback editor to make changes to the ADL-generated diagrams. The feedback portion of the editor implies that the developer should be able to add and remove icons and edges and have these changes reflected back into the specification database.

Other types of diagrams, for example, Rumbaugh's OMT and Yourdon's and Constantine's Structure Charts, should be modeled to reveal more aspects of general diagram layout in which only experimentation can reveal.

REFERENCES

[1] Batini, C., Furlani, L., and Nardelli, E., *What is a good diagram? A pragmatic approach*, Proceedings of the 4th Internat. Conf. on Entity-Relationship Approach (Silver Spring, MD, USA), IEEE Comput. Soc. Press, Oct. 1985, pp. 312-319.

[2] Belter, Stephen E., *Computer-aided Routing of Printed Circuit Boards*, BYTE, June 1987, pp. 199-208.

[3] D.A. Bocking, *The Attributed Translation System User's Guide*, Department of Computational Science, University of Saskatchewan, Saskatoon, December, 1984.

[4] Chen, P. P. *The Entity-Relationship Model - Toward a Unified View of Data*, ACM Transactions on Database Systems, Vol. 1, No. 1, pp. 9-36, March 1976.

[5] Devito, A., and MacKenzie, C., *Diagram Layout Language: A Placement Language for use in the Automatic Generation of Diagram Layouts*, University of Saskatchewan CMPT 833.3 & 429.3 Project Write-up, December, 1990.

[6] Di Battista, G., Eades, P., Tanassia, R., Tollis, I.G., *Algorithms for Drawing Graphs: an Annotated Bibliography*, Technical report No. CS-89-09, department of Computer Science, Brown University, October 1989.

[7] Fisher, C.N., Milton, D.R., and Quiring, S.B., *An Efficient Insertion-Only Error-Corrector For LL(1) Parsers*, Fourth ACM Symposium on the Principles of Programming Languages, 1977, pp. 97-103.

[8] Flemming, U., Rychener, M.D., Coyne, R.F., and Gavin, T.J. *A Generative Expert system for the Design of Building Layouts, Version 1: A Progress Report June 1986*, Pittsburgh: Carnegie-Mellon University, 1986.

[9] Gadwal, D., Findeisen, Piotr S. , Sorenson, Paul G. , Tremblay, J. Paul, Millar, L. Beth, *Generating Customizable Specification Environments Using Metaview*, Technical Report 94-2, Department of Computational Science, University of Saskatchewan, March 1994.

[10] Gane, C. and Sarson, T., *Structured Systems Analysis: tools and techniques*, Prentice-Hall, Englewood Cliffs, NJ, USA (1979).

[11] Gansner, E., North, S.C., and Vo, K.P., *DAG — A Program that Draws Directed Graphs*, Softw. Pract. Exper., 18, 11 (Nov 1988), pp. 1047-1062.

[12] Garey, M.R., Johnson, D.S. and Stockmeyer, L., *Some Simplified NP-complete Problems*, Proc. 6th annual ACM Symposium on the Theory of Computing, Seattle, April 1974, pp. 47-63.

[13] Hanan, M and Kurtzburg, J.M., *Placement Techniques in Design Automation of Digital Systems. Vol 1, Theory and Techniques*, Prentice-Hall, Englewood Cliffs, NJ, USA (1972), Chapter 5.

[14] Kundu, S. *The Equivalence of the Subregion Representation and the Wall Representation for a Certain for a certain Class of Rectangular Dissections*, Communications of the ACM, 31, 6, June 1988, pp. 752-763.

[15] Kusalik, Anthony J., and Oster, Gregory M., *ICOLA - A tool for drawing images from Declarative Picture Specifications*, Technical Report 93-2, Department of Computational Science, University of Saskatchewan, August 1993.

[16] Lee, C.Y., *An Algorithm for Path Connections and Its Applications*, IRE Transactions on Electronic Computers, September 1961, pp. 346-365.

[17] Mallgren, W.R. *Formal Specification of Graphic Data Types*, ACM Transactions on Programming Languages and Systems, 4, 4, October 1982, pp. 687-710.

[18] A.J. McAllister, *Modeling Concepts for Specification Environments*, Ph.D. Thesis, Department of Computational Science, University of Saskatchewan, Saskatoon, 1988.

[19] Mitchell, W.J., *Formal Representations: A Foundation for Computer-aided Architectural Design*, Environment and Planning B: Planning and Design, 13, 2, April 1986, pp.133-162.

[20] Protsko, L. Beth, Sorenson, Paul G. , Tremblay, J. Paul, and Schaefer, Douglas A., *Towards the Automatic Generation of Software Diagrams*, IEEE Transactions on Software Engineering, Vol. 17, No. 1, January 1991, pp. 10-21.

[21] Raghavan, R.,Cahoon, J. and Sahni, S., *Manhattan and Rectilinear Wiring*, Technical Report 81-5, University of Minnesota, March, 1981.

[22] Rich, E., Knight, K., *Artificial Intelligence*, Second Edition, McGraw-Hill Inc., 1991.

[23] Sorenson, P.G., Tremblay, J.P., and McAllister, A.J. *The Metaview System for Many Specification Environments*, IEEE Software Vol. 5, No. 2 (March 1988), pp. 30-38.

[24] Tamassia, R., Batini C., and Talamo, M., *An Algorithm for Automatic Layout of Entity-Relationship Diagrams*, Entity-Relationship Approach to Software Engineering: Proceedings of the Third Int. Conference on Entity- Relationship Approach, Amsterdam/New York: North-Holland 1983, pp. 421-439.

[25] Tamassia, R., Di Battista, G., and Batini C., *Automatic Graph Drawing and Readability of Diagrams*, IEEE Transactions on Systems, Man, and Cybernetics, Vol. 18, No. 1, Jan/Feb 1988, pp. 61-79.

[26] Steven L. Tanimoto, *The Elements of Artificial Intelligence*, Computer Science Press, Rockville, Maryland, 1987, pp. 148-164.

[27] J.P. Tremblay and P.G. Sorenson, *The Theory and Practice of Compiler Writing*, McGraw-Hill Book Company, 1985.

[28] J.P. Tremblay and P.G. Sorenson, *An Implementation Guide To Compiler Writing*, McGraw-Hill Book Company, 1982.

[29] Edward Yourdon and Larry L. Constantine, *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*, Prentice-Hall, INC., Englewood Cliffs, New Jersey, 1979.

APPENDIX A

DLL Attributed Input Grammar

The following appendix contains the attributed translation grammar for the Diagram Layout Language (DLL) which was used as the input to the Attributed Translation System (ATS). ATS accepts formal production rules for an LL(1) grammar in which non-terminals are lower case, terminals are upper case, action routines are preceded by the "@" symbol, and attributes are preceded by either the "<", or ">" symbols for inherited, and synthesized attributes, respectively. Refer to [3] for a more detailed explanation of the ATS input syntax. Each production in this input grammar is assigned a production number.

A DLL program has four main sections, as outlined in Section 4.4. The *Diagram Header* section, the *Graphics Definition* section, the *Metaview Connection* section and the *Icon Placement* section. In production 1 below these sections can be seen. The *program_name*, *diagram_name_declaration*, *orientation_declaration* and the *standard_declaration* are all productions involved in the *Diagram Header* section. The *icon_type_declaration* and the *arc_type_declaration* are productions involved in the *Graphics Definition* section. the *mapping_declaration* producion is involved in the *Metaview Connection* section. The last production, *placement_rules*, is involved in the *Icon Placement* section. All of these production, of course, decompose into further, more refined, productions, but the main components of a DLL program are illustrated in production 1.

```
1. program:
    @init program_name>f,n
    diagram_type_name_declaration<f @init_htab_ctr
    orientation_declaration<f
    standard_declaration<f @init_itab_ctr
    icon_type_declaration<f @init_etab_ctr
    arc_type_declaration<f
    mapping_declaration<f @init_ptab_ctr
    placement_rules<f @init_rtab_ctr
    END @close_output_file<f @reset @cleanup;
```

Diagram Header Section:

```
2.   program_name>f,n:
       PROGRAM COLON program_name_ident>n @set
         @open_output_file<n>f SEMI;

3.   program_name_ident>n:
       IDENT>n;

4.   diagram_type_name_declaration<f:
       | DIAGRAM_TYPE_NAME>d COLON IDENT>x @init_header_table<d,x,f SEMI;

5.   orientation_declaration<a:
       | ORIENTATION>f COLON orientation>o @init_header_table<f,o,a SEMI;

6.   orientation>o:
       TOP_DOWN>o |
       LEFT_RIGHT>o |
       NOT_FIXED>o;
```

```
7.   standard_declaration<a:
       | STANDARD>f COLON standard>s @init_header_table<f,s,a SEMI;

8.   standard>s:
       GRID>s |
       DIAGONAL>s |
       CURVED>s |
       MIXED>s;
```

Graphics Definition Section:

```
9.   icon_type_declaration<f:
       | ICON_TYPES COLON icon_type_list<f SEMI;

10.  icon_type_list<f:
       LB icon_ident_list<f rest_of_icon_ident_list<f RB;

11.  rest_of_icon_ident_list<f:
       | COMMA icon_ident_list<f rest_of_icon_ident_list<f;

12.  icon_ident_list<f:
       label_status>l LP IDENT>x @verify_icon_lib<x @init_icon_table<x,l,f
         rest_of_icon_list<f,l RP;

13.  rest_of_icon_list<f,l:
       | COMMA IDENT>x @verify_icon_lib<x @init_icon_table<x,l,f
           rest_of_icon_list<f,l;

14.  arc_type_declaration<f:
       | ARC_TYPES COLON arc_type_defn_list<f SEMI;
```

```
15.  arc_type_defn_list<f:
       LB arc_type_definition<f rest_of_arc_type_definitions<f RB;

16.  rest_of_arc_type_definitions<f:
       | COMMA arc_type_definition<f rest_of_arc_type_definitions<f;

17.  arc_type_definition<f:
       label_status>l direction>d LP arc_type>x @init_edge_table<x,l,d,f
         arc_type_definitions<l,d,f RP;

18.  arc_type_definitions<l,d,f:
       | COMMA arc_type>x @init_edge_table<x,l,d,f arc_type_definitions<l,d,f;

19.  label_status>l:
       LABELLED>l |
       NON_LABELLED>l;

20.  direction>d:
       UNI_DIRECTIONAL>d |
       BI_DIRECTIONAL>d |
       NON_DIRECTED>d;

21.  arc_type>x:
       IDENT>x;
```

Metaview Connection Section:

```
22.  mapping_declaration<f:
       | mapping<f mappings<f;

23.  mapping<f:
```

```
        MAP_ENTITY map_entity_tail<f SEMI |
        MAP_REL map_rel_tail<f SEMI;

24.  mappings<f:
        | mapping<f mappings<f;

25.  map_entity_tail<f:
        IDENT>a TO IDENT>x label>y @update_icon_table<a,x,y,f;

26.  label>y:
        @syn_NULL>y
        | LABEL IS LB IDENT>y RB;

27.  map_rel_tail<f:
        mr_arc_type>a TO IDENT>x
        SENDER IS ROLE NUMBER>n RECEIVER IS ROLE NUMBER>m
        label>l @update_edge_table<a,x,n,m,l,f;

28.  mr_arc_type>x:
        IDENT>x;
```

Icon Placement Section:

```
29.  placement_rules<f:
        @inh_to_NULL<f
        | rule_block<f rule_blocks<f;

30.  rule_blocks<f:
        @inh_to_NULL<f
        | rule_block<f rule_blocks<f;
```

```
31.  rule_block<f:
       RULE_BLOCK @set icon_entity_association abs_placement_rules<f
         rel_placement_rules<f conflict_resolutions @reset ENDBLOCK;

32.  icon_entity_association:
       entity_with_icon entities_with_icons;

33.  entities_with_icons:
       | entity_with_icon entities_with_icons;

34.  entity_with_icon:
       ASSOCIATE IDENT>z @insert<z COLON IDENT>x @retrieve<z>y
         @update_ent<y,x SEMI;

35.  abs_placement_rules<f:
       @inh_to_NULL<f
       | special_placement<f abs_placement_rules<f;

36.  rel_placement_rules<f:
       @inh_to_NULL<f
       | rule<f rel_placement_rules<f;

37.  rule<f:
       case_statement<f |
       grouping SEMI;

38.  special_placement<f:
       group_placement>t LP IDENT>x  RP SEMI |
       icon_diagram_placement>t LP IDENT>x RP SEMI |
       placement_property>t LP IDENT>x RP SEMI;
```

```
39. case_statement<f:
      RELATIONSHIP arc_type2>y,a,b cases<y,f ENDRELATIONSHIP;

40. arc_type2>y,a,b:
      IDENT>x LP simple_ident_or_uscore_list>a,b RP;

41. cases<y,f:
      @inh_to_NULL<y @inh_to_NULL<f
      | case<y,f cases<y,f;

42. case<y,f:
      CASE simple_ident_or_uscore_list>a,b COLON
        placement_statement_list<y,f,a,b ENDCASE;

43. simple_ident_or_uscore_list>a,b:
      IDENT>x @retrieve<x>a COMMA ident_or_uscore>b
      | USCORE @syn_uscore>a COMMA ident_or_uscore>b;

44. ident_or_uscore>b:
      IDENT>x @retrieve<x>b
      | USCORE @syn_uscore>b;

45. conflict_resolutions<f:
      | conflict_resolution<f conflict_resolutions<f;

46. conflict_resolution<f:
      CONFLICT COLON
      LB
        edge_ident_list<f
```

```
    RB
    RESOLUTION_MAPPING COLON
    LB
      res_mapping_statements<f
    RB;

47. edge_ident_list<f:
    IDENT>a @clr_et_part_ctr @clr_et_ctr
      LP icon_types COMMA @incr_et_part_ctr @clr_et_ctr
      icon_types RP @incr_et_part_ctr
      @clr_conflicting_rel_tab_ctr @get_edge_types<a,f COMMA
    IDENT>e @clr_et_part_ctr @clr_et_ctr
      LP icon_types COMMA @incr_et_part_ctr @clr_et_ctr
      icon_types RP @incr_et_part_ctr @get_edge_types<e,f
    rest_of_edge_list<f SEMI;

48. rest_of_edge_list<f:
    | COMMA IDENT>x @clr_et_part_ctr @clr_et_ctr
        LP icon_types COMMA @incr_et_part_ctr @clr_et_ctr
        icon_types RP @incr_et_part_ctr @get_edge_types<x,f
      rest_of_edge_list<f;

49. icon_types:
    IDENT>x @retrieve<x>y @update_et<y icon_type;

50. icon_type:
    | BAR IDENT>x @retrieve<x>y @update_et<y icon_type;

51. res_mapping_statements<f:
    | res_mapping_statement<f res_mapping_statements<f;
```

```
52. res_mapping_statement<f:
      simple_mapping_expr<f COLON PRIORITY_IS simple_res_expr<f SEMI;

53. simple_mapping_expr<f:
      @clr_conflict_resolution_tab_ctr mapping_term<f;

54. mapping_term<f:
      mapping_factor<f mapping_term_phrase<f;

55. mapping_term_phrase<f:
      | AND mapping_factor<f mapping_term_phrase<f;

56. mapping_factor<f:
      IDENT>x LP participant>t RP EQ IDENT>z LP role>u RP
        @init_con_res_tab<x,t,z,u,f;

57. role>t:
      SENDER>t
      | RECEIVER>t;

58. simple_res_expr<f:
      res_term<f;

59. res_term<f:
      res_factor<f res_term_phrase<f;

60. res_term_phrase<f:
      | AND res_factor<f res_term_phrase<f;
```

```
61.  res_factor<f:
       IDENT>x LP participant>t RP
         @update_con_res_tab<x,t,f;

62.  placement_statement_list<y,f,a,b:
       simple_placement_expr<y,f,a,b  SEMI statements<y,f,a,b;

63.  statements<y,f,a,b:
       | simple_placement_expr<y,f,a,b SEMI statements<y,f,a,b;

64.  simple_placement_expr<y,f,a,b:
       placement_term<y,f,a,b;

65.  placement_term<y,f,a,b:
       placement_factor<y,f,a,b placement_term_phrase<y,f,a,b;

66.  placement_term_phrase<y,f,a,b:
       | AND placement_factor<y,f,a,b placement_term_phrase<y,f,a,b;

67.  placement_factor<y,f,a,b:
       placement_definition<y,f,a,b
       | grouping;

68.  placement_definition<y,f,a,b:
       icon_icon_or_group_placement<y,f,a,b;

69.  icon_icon_or_group_placement<y,f,a,b:
       icon_icon_placement>t participants>h,i @update_icon_placement_table<y,t,a,b,f,h,i
       | group_placement>t LP IDENT>x RP;
```

```
70.  participants>x,a:
       | LP ident_or_uscore>x COMMA ident_or_uscore>a RP;

71.  icon_icon_placement>t:
       ABOVE>t |
       BELOW>t |
       LEFT_OF>t |
       RIGHT_OF>t |
       ABOVE_INDENT>t |
       BELOW_INDENT>t |
       DIRECTLY_ABOVE>t |
       DIRECTLY_BELOW>t |
       DIRECTLY_LEFT_OF>t |
       DIRECTLY_RIGHT_OF>t |
       CENTERED_ABOVE>t |
       CENTERED_BELOW>t |
       CENTERED_LEFT_OF>t |
       CENTERED_RIGHT_OF>t |
       NEXT_TO>t;

72.  grouping:
       @group_not_imp GROUP IDENT>x @insert<x COLON LB ident_list RB;

73.  ident_list:
       IDENT>x @insert<x rest_of_list;

74.  rest_of_list:
       | COMMA IDENT>x @insert<x rest_of_list;

75.  group_placement>t:
```

```
      IN_HORIZ_LINE>t
      | IN_VERT_LINE>t;

76. icon_diagram_placement>t:
      CONTAINED_IN>t |
      EXTERIOR_TO>t |
      CENTER_OF>t |
      TOP_CENTER_OF>t |
      LEFT_CENTER_OF>t |
      TOP_LEFT_OF>t |
      ON_BORDER_OF>t;

77. placement_property>t:
      REPLICATED_IN>t |
      UNIQUE_IN>t;

78. if_statement:
      IF bool_factor>y,a,b THEN ELSE ENDIF;

79. bool_factor>y,a,b:
      arc_type2>y,a,b;
```

APPENDIX B

Questionnaire

The following five diagrams are the output of the newly-developed Automatic Diagram Layout (ADL) System prototype. Two diagram *types* were chosen to be modeled by the ADL System. The Two diagram types are:

1. Data Flow Diagrams
2. Entity-Relationship Diagrams

There are three Data Flow diagrams: Library System, First Level DFD for a Customer Ordering System, and a Second Level DFD for the Customer Ordering System (blow-up of the Makeup Sales Record process of the Customer Ordering System). There are two Entity Relationship (ER) Diagrams: A Book Classification ER Diagram, and an Airport ER.

Questions

1. Rate on a a scale of 1 to 4 (1 = poor, 2 = fair, 3 = good, 4 = excellent) each diagram according to the following criteria:

Criteria	Explanation of Criteria
AREA	Does the diagram occupy an appropriate amount of space?
BALANCE	Does the diagram occupy its space reasonably? (lopsided?, uneven?, asymmetrical?)
BENDS	Are the bends in the arcs on the diagram appropriate?
DIMENSION	Do icons of the same type differ in size in such a way as to distract from their meaning or importance on the diagram?
LENGTH	Are the lengths of the arcs on the diagram appropriate?
ICON LABELS	Are the labels for icons meaningful and placed well?
ARC LABELS	Are the labels for arcs meaningful and placed well?
OVERALL	Give an overall rating of the diagram.

Please use the following table for your objective evaluation: (Note: D1 means Diagram 1)

Criteria	D1	D2	D3	D4	D5
AREA					
BALANCE					
BENDS					
DIMENSION					
LENGTH					
ICON LABELS					
ARC LABELS					
OVERALL					

2. As a software developer, would you consider some, all, or none of the diagrams acceptable as a first draft from an automatic diagram layout tool knowing that you could edit it afterwards? Please give your reasons.

3. What are your overall impressions of the diagrams generated by the ADL System prototype? For example, what are the diagrams' strengths and weaknesses?