From Formal Specifications to Natural Language: A Case Study

J. M. Punshon & J. P. Tremblay

Department of Computer Science
University of Saskatchewan
Saskatoon, Canada S7N 5A9
Ph: 306-966-4908
Fax: 306-966-4884
Email: tremblay@cs.usask.ca

P. G. Sorenson & P. S. Findeisen

Department of Computing Science
University of Alberta
Edmonton, Canada T6G 2H1
Ph: 403-492-4589
Fax: 403-492-1071
Email: sorenson@cs.ualberta.ca

Abstract

Formal descriptions, while difficult for most human readers to understand, are convenient for specifying large software systems, where completeness and consistency are important issues. Informal specifications can offer advantages in readability, but ambiguities and contradictions are unavoidable side-effects. Since a specification often acts as a formal contract between the software developer and the customer, it is essential that both sides be able to fully understand the specification document. Systems have been proposed which help the software client better understand the specification by automatically paraphrasing it in natural language.

Metaview is a customizable metasystem that facilitates the construction of software specification environments that can be used to support software specification tasks. This paper describes a strategy for producing natural language descriptions of formal software requirements specifications developed using Metaview specification environments. A case study demonstrates, using the Object Modeling Technique specification methodology as an example, how the system generates a textual description from a database of software requirements.

1 Introduction

The production of a major software system is generally considered to be a very complex activity. In an attempt to deal with this complexity, the system development process is often decomposed into distinct phases that are generally identified as requirements analysis, system specification, system (architectural and detailed) design, coding, integration and testing, operation, and maintenance. Much of the current research in computer-aided software engineering (CASE) environments is focused on the early phases of development: requirements analysis, system specification, and system design. A primary motivation for this focus is the realization that defects are much more costly to fix if they are found in the later phases of system development.

1.1 Motivation

This paper deals with the requirements analysis, elicitation, and specification phase of development. In this stage a customer's statement of requirements is the input, and a formal specification of the system is produced. It is often advantageous to capture a specification in some formal notation, rather than describing it in natural language. CASE tools can automatically check formal specifications for consistency and some kinds of completeness; however, informal specifications are much easier for most people to understand. Even in situations where a CASE tool provides a graphical representation of the specification information, users who are unfamiliar with the graphical notation used can be aided by a companion textual description. For this reason, tools have been proposed which paraphrase a formal description in easy-to-read natural language text.

1.2 Related Work

In the past fifteen years, natural language generation has been applied to a number of specific subject areas, including the translation of formal software specifications into natural language. Swartout's GIST English Generator [14] produces descriptions of program specifications written in the GIST specification language. The ARIES (Acquisition of Requirements and Incremental Evolution of Specifications) component of the Knowledge-Based Software Assistant [6] is capable of presenting a given software specification in a variety of different styles, including graphical portrayals, formal descriptions, and natural language paraphrases. The GETS system [11] generates an English description of a specification written in Telos [8], an object-oriented requirements modeling language intended to support the development of information systems.

A summary of these efforts in the translation of formal software specifications into natural language can be found in [12].

1.3 Overview of Paper

Section 2 gives a brief review of the metasystem architecture for Metaview and its associated data model (EARA/GE) used in describing and producing specification environments. The techniques used to create object-oriented specification environments in Metaview are discussed and the approach is illustrated with the description of an environment to support OMT object modeling [10]. Metaview's ability to accurately capture requirements information based on this methodology is illustrated with an example in which an Automated Teller Machine (ATM) System described by Rumbaugh et al. [10] is modeled.

Section 3 focuses on the REVIEW system for transforming formal specifications into English. The design of the REVIEW system is described and illustrated using the OMT methodology and the ATM requirements specification as examples.

The OMT object model was chosen as the example for this exercise because it is a popular object-oriented technique and because the diversity of its three distinct models exemplifies the flexibility of the report generator. The complete OMT methodology, including object model, dynamic model, and functional model, is supported by Metaview (see [15]) but due to constraints on the length of this paper, we focus solely on the object model.

The final section of the paper summarizes and discusses future work.

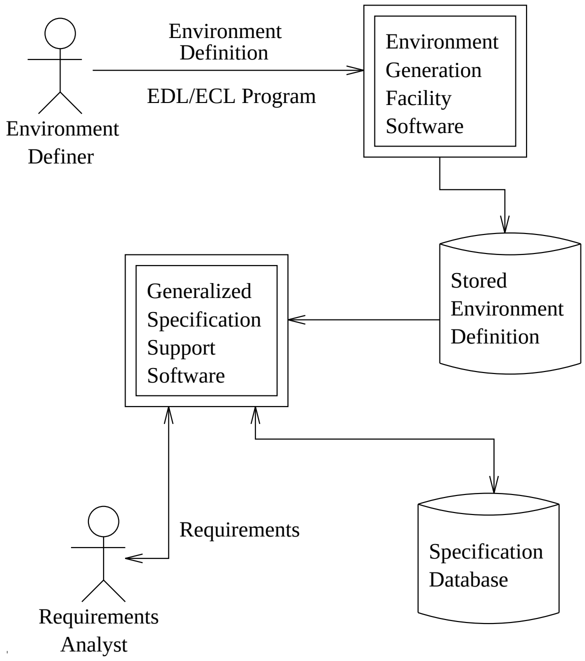


Figure 1: Simple Metaview organization.

2 Metaview Environments

The main goal of a metasystem is to generate automatically a large part of the software necessary for CASE environments [4]. This section gives a brief overview of the Metaview architecture. The underlying data model of Metaview, the EARA (Entity-Aggregate-Relationship-Attribute) model, and its GE (graphical extension), which is used to define the software specification objects, is then described. A sample specification environment to support the OMT Object Model is given, and a sample specification for an Automated Teller Machine system is presented.

2.1 Metasystem Architecture

In Metaview a generative approach to the creation of software specification environments using a metasystem is adopted [5, 9, 13]. In a generative approach, a particular software specification environment is generated from a formal definition (or model) of that environment. There are three levels of specification in most metasystems that use a generative approach: the *meta* level, the *environment* level, and the *user* level. The user in this case is the analyst/designer. Only the last two levels appear in Figure 1.

The *metasystem definer* defines the underlying model for the metasystem and creates the software to support the definition of specification environments using this model.

The *environment definer* uses the environment generation facility provided by the metasystem software to define a model that supports a particular specification methodology, such as Object Oriented Analysis. This model (called the *environment model*) defines the terminology that is to be used to specify requirements within that environment (e.g., classes, associations, and qualifiers in the OMT object model).

The *analyst* or *requirements engineer* uses the support software provided by the metasystem to help store and analyze one or more requirements specifications. The terms and concepts that the analyst uses are different depending on the environment used. The *Generalized Specification Support Software* shown in Figure 1 may include several different facilities for analyst interaction with the requirements database. Examples include a database engine, a textual batch input processor, interactive text and graphics processors, automatic layout generator system, and a formal specification to English generator system, which is the focus of this paper. These facilities are used to store, manipulate, and analyze specifications for any software system that analysts want to define. Each specification is stored in a requirements database.

2.2 EARA/GE Data Model

The meta-model defined for the Metaview system is called EARA/GE (Entity-Aggregate-Relationship-Attribute with Graphical Extension). As the name suggests, EARA is based on the entity-relationship data model [2], but incorporates significant extensions, like specialization, aggregation, and some elements of generalization.

The basic concepts used in the EARA model are *entities*, *aggregates*, *relationships*, and *attributes*. Entities represent real-world objects. Relationships are associations that exist between entities and/or aggregates. The entities and aggregates that take part in a relationship are called *participants* of that relationship. 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. *Aggregation* is a mechanism that allows an aggregate, which is composed of many entities and relationships, to be treated as a single identifiable entity within another aggregate. Aggregation is an indispensable feature in modeling various decomposition approaches used in requirement specification and design methodologies.

Entities, relationships and aggregates can have *attributes*. The attributes represent properties of the objects 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 has a type. The objects of the same type share many properties, like aggregation capabilities for entities, roles for relationships, eligible types of components for aggregates and the names and types of attributes. The object types can form *specialization hierarchies*. The hierarchy is formed when an object type is defined as a *subtype* of another type. Subtypes inherit all the properties of their supertypes.

In addition, a graphical extension (GE) to the EARA model has been designed [9] that supports the definition of a graphical representation for software objects definable in the basic EARA model. Metaview's meta-model has been used to model some twenty software methods, including structured analysis and design approaches, object-oriented approaches such as OMT, and other approaches such as ADDISSA and box notation.

2.3 OMT Object Model Environment

This section sketches the Object Modeling Technique (OMT), popularized by Rumbaugh et al. [10]. OMT proposes to develop software systems in four stages: analysis, system design, object design, and implementation. Each of these stages uses a development methodology composed of an *Object Model*, a *Dynamic Model*, and a *Functional Model*. In this paper we focus on the application of the Object Model in the analysis stage.

The Object Model describes the structure of object classes, with their various attributes and operations, and the various associations among them. Classes are represented graphically by boxes, and associations are represented by arcs linking the related classes. Binary and n-ary associations are permitted, and each participant in an association can be optionally named or given multiplicity. Several special associations exist, like specialization and aggregation. Their graphical representation is different from the regular associations and includes ornaments drawn on the arcs.

The conceptual contents of Object Model diagrams can be presented by the aggregate type *object_model*. To simplify the modeling through the use of subtyping, an abstract entity type called *om_entity* (object model entity) and an abstract relationship type called *om_relationship* are defined as the component types of the aggregate. These types will be supertypes for all entity and relationship types used within the Object Model. In Metaview environment definitions, the keyword GENERIC identifies a type which is permitted to have subtypes.

```
AGGREGATE_TYPE object_model
   COMPONENTS (om_entity, om_relationship);
ENTITY_TYPE om_entity GENERIC;
RELATIONSHIP_TYPE om_relationship GENERIC;
```

Central notions in OMT's Object Model are class and association (between or among classes). The classes can be represented by EARA entities; however, it is impossible for the associations to be represented by EARA relationships. This is because the associations in the Object Model can participate in other associations, while the EARA relationships can take only entities as participants. To reflect all dependencies from the Object Model, we map associations to entities. Thus, each binary association binding two classes in the Object Model is defined as an entity representing the association itself (type *binary_association*), and two relationships (of type *binding*) which bind the association entity to the corresponding class entities (type *class*). The *binding* relationship has three subtypes: *binds*, *binds_many*, and *binds_optional*, which reflect the possible cardinalities of the association with respect to the class object being the participant of the relationship.

```
ENTITY_TYPE class IS_A om_entity;
ENTITY_TYPE association GENERIC IS_A om_entity
  ATTRIBUTES (assoc_name : string);
ENTITY_TYPE binary_association IS_A association;
ENTITY_TYPE n_ary_association IS_A association;
ENTITY_TYPE role IS_A om_entity
  ATTRIBUTES (role_name: identifier);
```

```
RELATIONSHIP_TYPE binding GENERIC IS_A om_relationship;
  ROLES (association, role, participant)
  PARTICIPANTS (binary_association, role, class | qualifier);
RELATIONSHIP_TYPE binds IS_A binding
  PARTICIPANTS (n-ary_association, role, class);
RELATIONSHIP_TYPE binds_many IS_A binding
  ATTRIBUTES (multiplicity : string);
RELATIONSHIP_TYPE binds_optional IS_A binding;
```

The three roles defined for the *binding* relationship type (*association*, *role*, and *participant*) are inherited by all its subtypes. Similarly, the set of participants defined for this relationship type is also inherited. The vertical bar represents an alternative. Thus the possible combinations of participants for *binding* include a *binary_association* entity for the *association* role, a *role* entity for *role*, and a *class* or a *qualifier* for the third, participant role. An additional combination of participants, which includes an *n_ary_association*, is defined for the *binds* relationship type only.

One of the more complicated constructions in the Object Model is the qualified association. It involves two classes, an association, and a qualifier. The qualifier is a special attribute that is written in a small box at one end of the association line and adjacent to one of the classes. The qualifier can be represented by an entity, but its connections with the class and the association must be also modeled at the conceptual level. We do this by introducing the relationship type *qualification* that binds the qualifier and the class. Thus, the elements needed to build a qualified association have the following definition:

```
ENTITY_TYPE qualifier IS_A om_entity;
RELATIONSHIP_TYPE qualification IS_A om_relationship
  ROLES (class, qualifier)
  PARTICIPANTS (class, qualifier);
```

The Object Model's aggregation and specialization are modeled in a way very similar to the binary and n-ary associations.

```
ENTITY_TYPE part_of IS_A om_entity;

RELATIONSHIP_TYPE aggregation_from GENERIC IS_A om_relationship;
    ROLES (component, role, structure)
    PARTICIPANTS (class, role, part_of );
RELATIONSHIP_TYPE aggregate_from IS_A aggregation_from
RELATIONSHIP_TYPE aggregate_from_many IS_A aggregation_from
    ATTRIBUTES (multiplicity : string);
RELATIONSHIP_TYPE aggregate_from_optional IS_A aggregation_from;

RELATIONSHIP_TYPE aggregation_to GENERIC IS_A om_relationship;
    ROLES (structure, composite)
    PARTICIPANTS (part_of, class);
```

```
RELATIONSHIP_TYPE aggregate_to IS_A aggregation_to
RELATIONSHIP_TYPE aggregate_to_many IS_A aggregation_to
    ATTRIBUTES (multiplicity : string);
RELATIONSHIP_TYPE aggregate_to_optional IS_A aggregation_to;

ENTITY_TYPE is_a IS_A om_entity;

RELATIONSHIP_TYPE specialize_from IS_A om_relationship
    ROLES (superclass, structure)
    PARTICIPANTS (class, is_a);
RELATIONSHIP_TYPE specialize_to IS_A om_relationship
    ROLES (structure, subclass)
    PARTICIPANTS (is_a, class);
```

2.4 An Example OMT Requirements Database

This section illustrates an Object Model which captures the requirements for an ATM system. The example ATM system (shown in Figure 2) is taken from Rumbaugh et al. [10]. In Metaview, the requirements are specified by using a graphical editor. Here, however, we present the requirements in a formal textual language in order to give the reader a better idea of the precise contents of the requirements database.

A description of the information content of a specification database at a particular point in time must define the aggregates, entities, relationships, and associated attribute values that make up a database state. Those database "objects" are defined using the aggregate, entity, relationship, and attribute type names defined by a particular environment definition. Since the types names differ for each environment definition, the specification language must be defined to incorporate different "object" type names depending on the environment definition used. Thus there actually is a *class* of specification languages, one specification language for each environment definition. The specification languages differ only in terms of the "object" type names used. We refer to this family of specification languages as the Simple Specification Languages (SSL's) [7]. In this section we use the SSL for specifying OMT Object Models.

A partial SSL specification of the ATM system in Figure 2 appears in Figure 3. This example specification will be used in the next section to illustrate our approach to the generation of English text from a formal SSL specification.

3 Describing Requirements in Natural Language

This section introduces an enhanced version of the REVIEW system [12] for generating natural language descriptions of software requirements specifications. The modified approach described here gives REVIEW access to environment-specific configuration information, which increases the system's ability to interpret requirements captured in a wide range of specification environments.

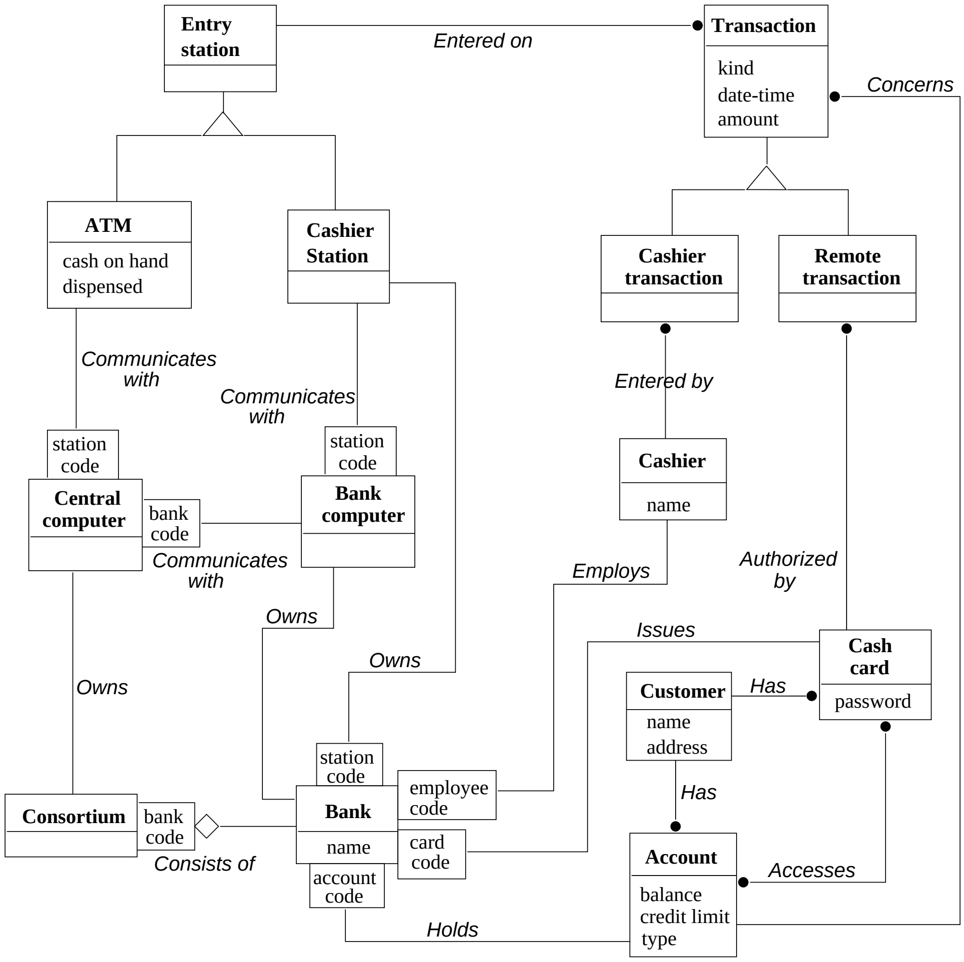


Figure 2: OMT object model diagram for ATM system [10].

```
BEGIN object_model atm_example

        class Entry_station;
        class ATM;
        class Cashier_station;
        class Transaction;
        class Central_computer;

        is_a is_a.0;
        binary_association binary_association.0
                (assoc_name = ‘entered on’);
        binary_association binary_association.3
                (assoc_name = ‘communicates with’);

        role role.0;
        role role.1
                (role_name = ‘subject’);
        role role.2
                (role_name = ‘subject’);
        role role.3;

        qualifier Station_code;

        binds (binary_association.0, role.1, Transaction);
        binds (binary_association.0, role.0, Entry_station);
        specialize_from (Entry_station, is_a.0);
        specialize_to (is_a.0, ATM);
        specialize_to (is_a.0, Cashier_station);
        binds (binary_association.3, role.3, ATM);
        binds (binary_association.3, role.2, Station_code);
        qualification (Central_computer, Station_code);

        .
        .
        .

END atm_example;
```

Figure 3: Partial SSL specification of the ATM system.

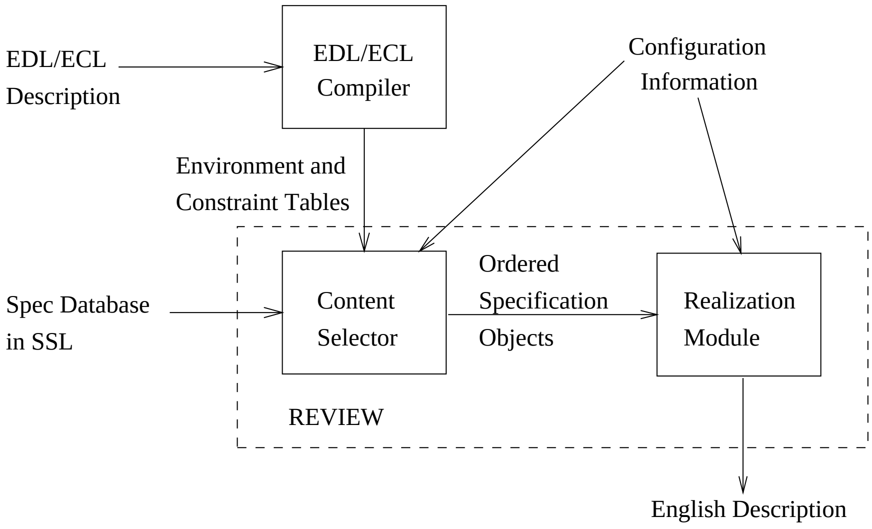


Figure 4: REVIEW architecture.

3.1 Goals of the REVIEW System

A report generated by REVIEW will only be of use if it helps a reader to understand the software requirements specification of the system or a subpart of the system in which the reader is interested.

It is essential that the generated report be coherent. REVIEW follows a procedure for building descriptive paragraphs based on Casty's model [1], in which a coherent structure is achieved through the use of sentences with parallel patterns, verb tense is consistently maintained, and the sentence subject shifts through a hierarchy from abstract to concrete and from general to specific and detailed. A concise organization is also important to aid readability [3]. A series of short, simple sentences quickly becomes tiresome to the reader, who is accustomed to a style in which related ideas are combined in a single sentence.

REVIEW's task is to plan a series of sentences which achieves a coherent, concise, and accurate description of the contents of a software requirements database. The generality of the approach used by REVIEW means that the requirements may be expressed in any methodology supported by Metaview.

3.2 REVIEW Architecture

An overview of the REVIEW system is shown in Figure 4. The Metaview EDL/ECL compiler produces environment tables which are used to instantiate the text generator for a particular environment. After this point, the REVIEW generator is detached from the rest of the Metaview system; it acts independently as a tool specific to the environment the analyst has chosen.

The requirements information (the SSL program) is captured in a specification database, which, along with some additional configuration information, forms the input to REVIEW. The output is an English text description of the database contents.

Generating a report is a two-stage process. First, the *content selector* chooses appropriate objects from the specification database and orders them in a specific fashion. These objects are then sent to the *realization module*, which generates sentences. Both the content selector and the realization module require additional input in the form of configuration information to help them interpret the software requirements.

3.3 Planning the Text

The overall planning strategy for a report generated by REVIEW is to describe the contents of the requirements database by focusing on each specification object in turn. Thus the content selector must choose an ordering for the specification objects that compose the requirements specification.

The content selector extracts the content from the specification database for each individual sentence required. This is accomplished through the use of a three-step graph traversal approach. The first step is to generate a graph representation of the database contents; and then apply graph reduction rules to simplify the graph. Finally, a graph traversal is performed to obtain a suitable ordering for the specification objects.

3.3.1 Graph Generation

A graph is generated for each aggregate object in the database. For example, the object model representing the ATM system shown in Figure 2 is a single aggregate, which contains many entities and many relationships among those entities.

There is a one-to-one correspondence between the aggregate's entities and the nodes of the graph for that aggregate. Each binary relationship is represented as an arc from the relationship's first participant to its second participant. Figure 5 shows a portion of the ATM system in three different forms: the object model diagram, the textual list of entities and relationships in the database, and the graph representation of the database contents. Several of the modeled OMT objects have a graphical representation but are not assigned any name by the requirements definer. The *is_a* centers are an example of this. When such an object is created under Metaview, a unique internal name is automatically assigned. In Figure 5b we see that the unnamed *is_a* center in Figure 5a is assigned the name *is_a.0*.

With traditional graph theory we are limited to arcs which connect only two nodes. However, information from the specification database also contains ternary relationships (which are infrequently encountered) and even relationships with four or more participants (which are extremely rare). We introduce the notation shown in Figure 6 for graphically representing a single relationship (an arc) which links more than two entities (nodes). Thus our graphs contain two distinct types of arcs; the first is the usual arc associated with graph theory, and the second is a single arc that can link more than two nodes.

The complete graph representation for the ATM example is reasonably large when shown in full. A portion of the ATM object model appears in Figure 7.

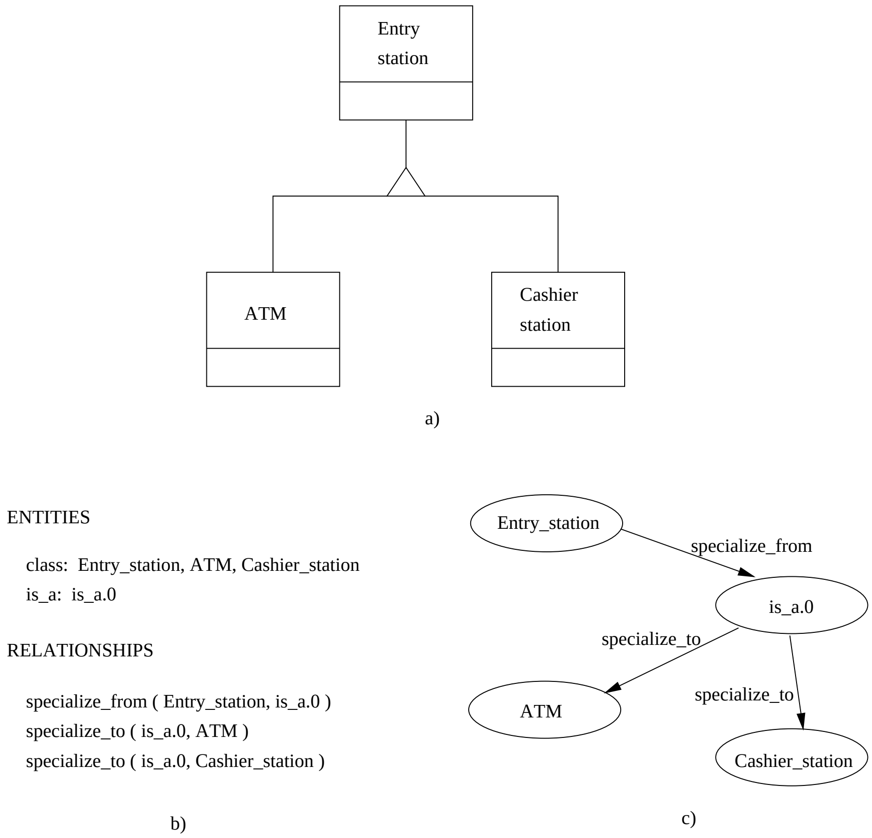


Figure 5: Graphical representation of specification information.

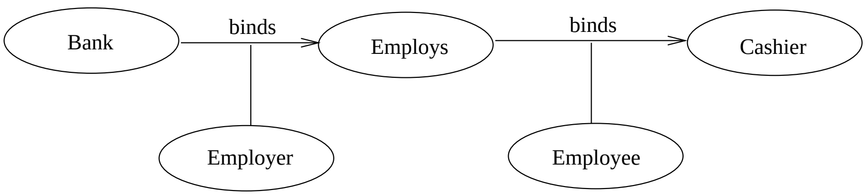


Figure 6: Graphical representation of ternary relationships.

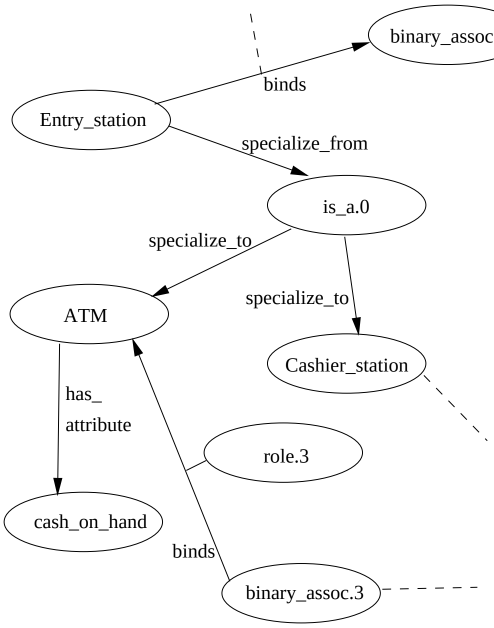


Figure 7: Graph representation of the ATM object model.

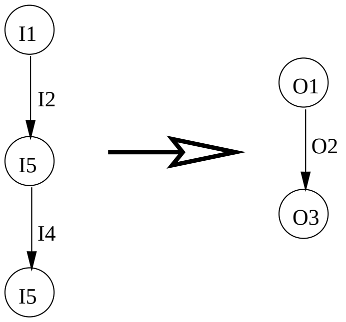
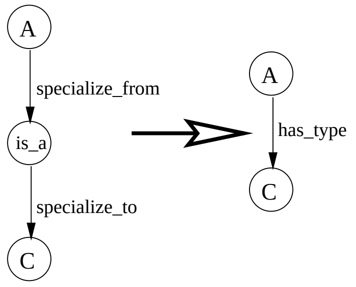


Figure 8: The Type I Reduction Rule

3.3.2 Graph Reduction

The graph reduction step applies environment-specific rules that remove unnecessary detail from the graph. These rules must be supplied by the environment definer, and their purpose is to aid REVIEW in interpreting the specification.

Recall that in modeling OMT in Metaview, we map OMT associations not as relationships but as entities. Thus the concept of specialization, in which one class is a subtype of another class, is represented in the specification database as the construct shown in Figure 5c. In this case we would like REVIEW to know that although three entities and the associated bindings are in the specification database, conceptually a relationship among two entities is intended. A graph reduction rule like the one shown in Figure 8 is required. This rule reduces a three-entity construct linked by two binary relationships to a two-entity construct with a single binary relationship. To facilitate the task of the environment definer who must supply the reduction rules, templates for




```
TypeI (
     ( A, 'specialize_from', 'is_a', 'specialize_to', C )
     ( A, 'has_type', C ) )
```

Figure 9: An instantiation of a Type I Reduction Rule

common rules are built-in to REVIEW. The built-in rules support the common notations used in Metaview to model specifiction environment concepts. Rules such as the one shown in Figure 8 are designated as Type I Reduction Rules.

Figure 9 shows an instantiation of the Type I template that an environment definer would use to deal with specialization in the object model. The reduction rules are the "background knowledge" that REVIEW needs to understand what the specification information means. In this case a node A is related to an *is_a* node by the *specialize_from* relationship, and a *specialize_to* relates that *is_a* node to another node C. In the OMT Object Model this is interpreted as C being a specialization, or subtype, of A.

The aggregation constructs used in the object model can also be reduced with Type I rules. Figure 10 shows some of the rules required; the other aggregation rules are similar.

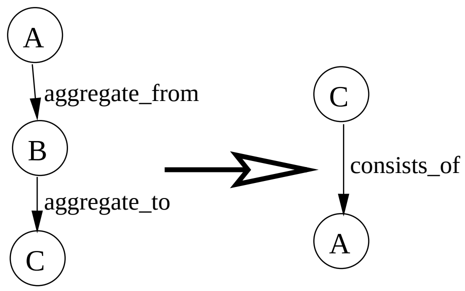
A Type II Reduction Rule (Figure 11) is used to handle binary associations. The association itself, represented as node B in the graph, becomes a directed arc. A direction for the arc must be determined from the information in the specification database, and it is the association's *role names* (nodes D and E) which give us the necessary clues. The rule, as it is stated, assumes that a role will be assigned for the entity A, but that none is supplied for entity C. The keyword NONE ensures that the rule is only applied in cases where node C's role name is left unspecified.

The graph representation of the ATM system's object model, after reduction rules have been applied, is shown in Figure 12.

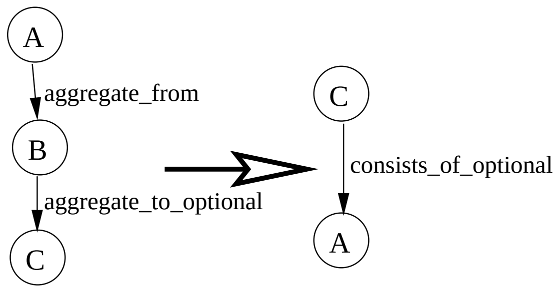
3.3.3 Graph Traversal

The task of determining an ordering for the specification information is accomplished by a graph traversal algorithm. This algorithm creates subgraphs from clusters of entities and relationships in the graph representation. The subgraphs are then passed on to the text realization module to be converted into English sentences.

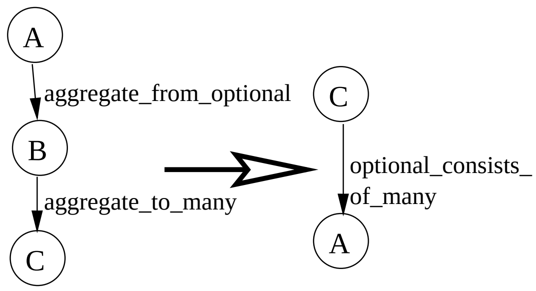
The choice of a starting node for the traversal is an important one. The directional arrows on the arcs of Figure 12 indicate the order in which information should be presented. That is, the mention of a given object will lead to the mention of its subtypes, thus the supertype should be described first. Similarly, *active* participants in a relationship should be described before their



TypeI (
(A, ‘aggregate_from’, B, ‘aggregate_to’, C)
(C, ‘consists_of’, A))



TypeI (
(A, ‘aggregate_from’, B, ‘aggregate_to_optional’, C)
(C, ‘consists_of_optional’, A))



TypeI (
(A, ‘aggregate_from_optional’, B, ‘aggregate_to_many’, C)
(C, ‘optional_consists_of_many’, A))

Figure 10: Some Type I Reduction Rule instantiations for aggregation.

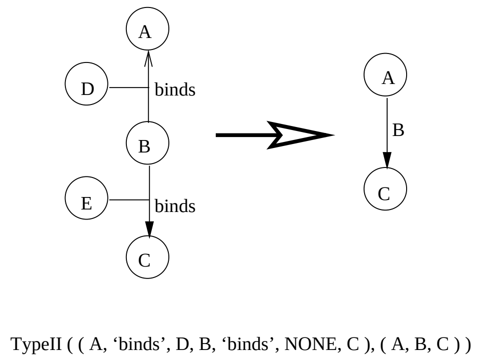


Figure 11: Type II Reduction Rule

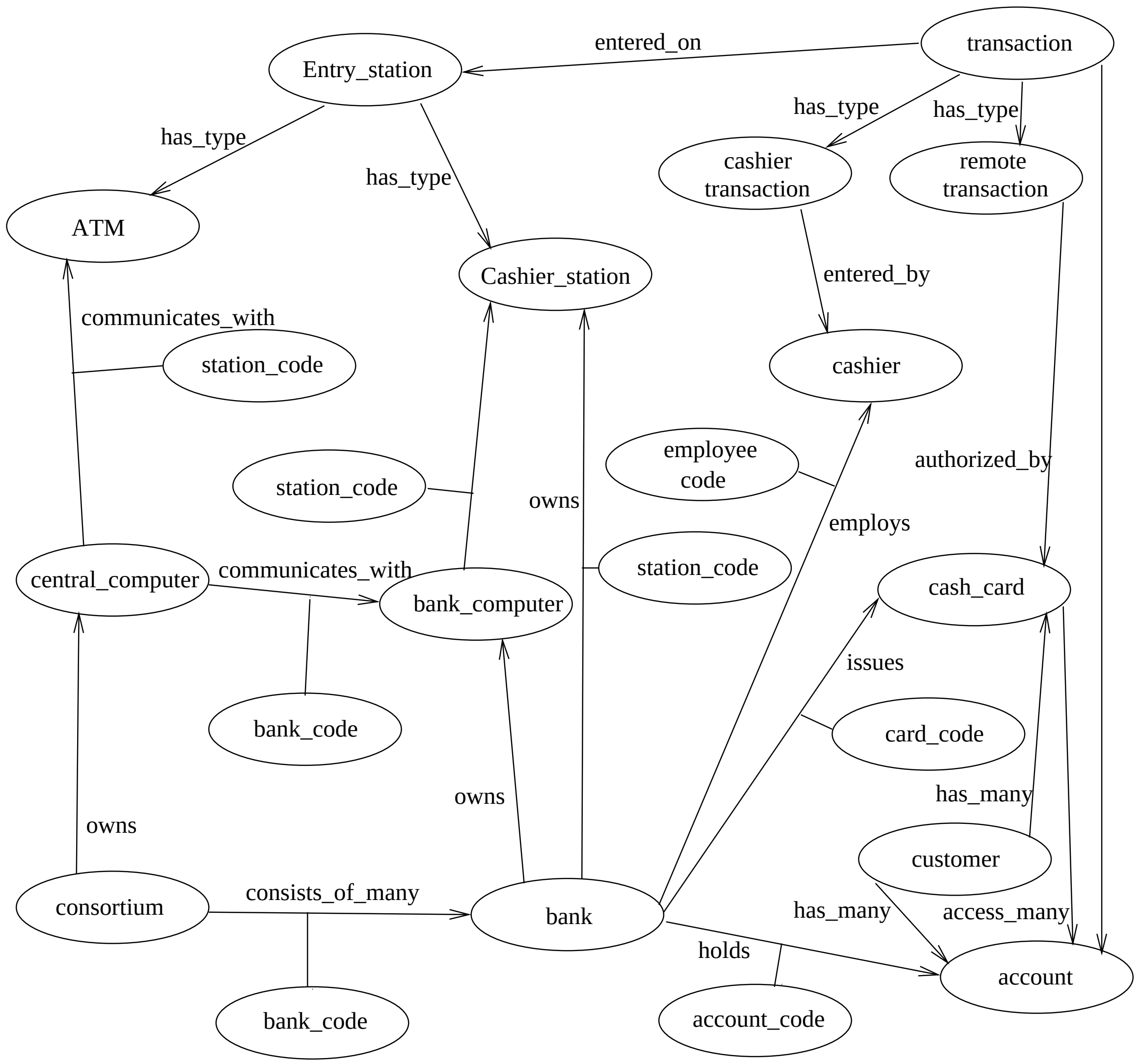


Figure 12: Reduced graph representation of ATM object model.

```
Algorithm Graph_Traversal

  declarations
      type Node:
          Priority: integer
          Visited: boolean
          end
      N : Node
      Start_Nodes : set of Nodes

  Start_Nodes <- each node that has no parents
  for each node N in Start_Nodes
      Prioritize (N)
  for each node N in Start_Nodes
      Traverse (N)
```

```
Algorithm Prioritize (N : Node)

  declataration D : Node
  if N.Visited = false
  then
      N.Visited <- true
      N.Priority <- N.Priority + 1
      for each neighbour D of N
          Prioritize (D)
      N.Visited <- false
```

```
Algorithm Traverse (N : Node)

  Declaration C: Node
  if N.Visited = false
  then
      N.Visited <- true
      K <- neighbours of N, sorted by Priority
      for each C in K
          Output subgraph with C, N, relationship(C,N)
          Traverse (C)
```

Figure 13: Graph traversal algorithm.

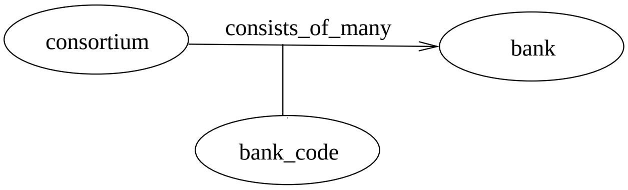
passive counterparts. For example, *The consortium consists of banks, which own bank computers, which communicate with cashier stations* is preferable to *a bank computer communicates with cashier stations, a bank owns a bank computer, and the consortium consists of banks.*

In Figure 12 there are three nodes which have eminating arcs only. These nodes, *transaction*, *customer*, and *consortium*, are the start nodes; a traversal will begin at each of these three points.

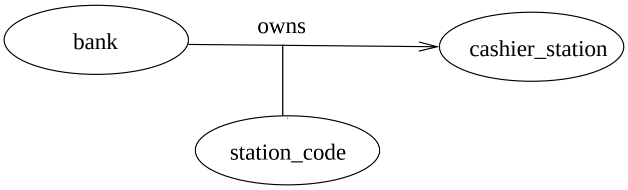
Traversal of the graph is depth-first but is done in two stages. An initial pass assigns integer priorities to the nodes (the *prioritize* pass) and the final pass uses these priority values to choose among multiple paths from a node (the *traversal* pass). A high-level algorithm is shown in Figure 13.

The prioritizing pass performs a depth-first traversal of the graph and assigns a higher priority to nodes which are reachable from the root by more than one path. An example of this is the node *cashier_station* in Figure 12, which is reachable from the node *consortium* by three distinct paths. By following the assigned priorities in the final traversal, the cluster of nodes involving *consortium*, *central_computer*, *bank*, *bank_computer*, and *cashier_station* will be kept adjacent in the content ordering. The content selector will then move on to other nodes.

The final traversal is done in a depth-first manner with the neighbours of the current node examined in an order determined by their priority ranking (highest priorities first). Because the graph may potentially contain loops, each node is marked *visited* as it is examined. The output of this algorithm is a series of subgraphs, each one based on a single relationship among two or more entities. From the start node *consortium*, subgraphs are constructed based on the following list of relationships:



consists_of_many (consortium, bank_code, bank)



owns (bank, station_code, cashier_station)

Figure 14: Subgraphs in the ATM object model.

1. consists_of_many (consortium, bank_code, bank)
2. owns (bank, station_code, cashier_station)
3. owns (bank, bank_computer)
4. communicates_with (bank_computer, station_code, cashier_station)
5. employs (bank, employee_code, cashier)
6. holds (bank, account_code, account)
7. issues (bank, card_code, cash_card)
8. access_many (cash_card, account)
9. owns (consortium, central_computer)
10. communicates_with (central_computer, bank_code, bank_computer)
11. communicates_with (central_computer, station_code, atm)

The subgraphs corresponding to the first two relationships in this list are shown in Figure 14. A series of subgraphs is produced from the other two graph roots in a similar manner.

3.4 Realizing the Text

Once the ordering has been determined by the content selector, the procedure for creating a report is to generate sentences for the individual subgraphs. The realization module by default treats the relationships as verbs, and the relationship's participants as nouns filling the roles of subject and objects. For the object model an additional rule is necessary to assert that the middle participant of a ternary relationship should be interpreted as a qualification attribute by which the first participant (the subject) identifies the third (the object). The result of this role interpretation is a functional description of a single sentence that identifies the sentence constituents and the roles they play. The series of subgraphs created by the content selector is used to generate the following list of sentences:

1. The consortium consists of banks (identified by a unique bank code).
2. A bank owns cashier stations (identified by a unique station code).
3. A bank owns a bank computer.
4. A bank computer communicates with cashier stations (identified by a unique station code).
5. A bank employs cashiers (identified by a unique employee code).
6. A bank holds accounts (identified by a unique account code).
7. A bank issues cash cards (identified by a unique card code).
8. Cash cards access accounts.
9. The consortium owns a central computer.
10. The central computer communicates with bank computers (identified by a unique bank code).
11. The central computer communicates with ATMs (identified by a unique station code).
12. Transactions are entered on an entry station.
13. An entry station may be an ATM or a cashier station.
14. Transactions concern accounts.
15. A transaction may be a cashier transaction or a remote transaction.
16. Remote transactions are authorized by a cash card.
17. Cashier transactions are entered by a cashier.
18. A customer has accounts.
19. A customer has cash cards.

When the realization module creates the functional descriptions of the second and third sentences, it recognizes that these two adjacent sentences have the same subject (*bank*) performing the same action (*owns*), so it combines the two into a single sentence: *A bank owns cashier stations (identified by a unique station code) and a bank computer.*

In the next subgraph, the subject *bank_computer* is the same as the object of the preceding sentence. The realization module combines the sentences in cases such as this. The following excerpt from a final report shows the generated description of the series of subgraphs starting at the *consortium* node.

The consortium consists of banks (identified by a unique bank code). A bank owns cashier stations (identified by a unique station code) and a bank computer, which communicates with cashier stations (identified by a unique station code). A bank employs cashiers (identified by a unique employee code) and holds accounts (identified by a unique account code). A bank issues cash cards (identified by a unique card code), which access accounts. The consortium owns a central computer, which communicates with bank computers (identified by a unique bank code) and with ATMs (identified by a unique station code). Transactions are entered on an entry station, which may be an ATM or a cashier station. Transactions concern accounts. A transaction may be either a cashier transaction or a remote transaction. Remote transactions are authorized by a cash card. Cashier transactions are entered by a cashier. A customer has accounts and cash cards.

4 Summary and Future Work

This paper has focused on the specification, analysis, and elicitation phase of software development. Some related work in transforming formal specifications to natural language was briefly mentioned. An overview of the Metaview metasystem architecture was given, including the definition of an OMT Object Model environment and a requirements database for an ATM system. Finally, the REVIEW system architecture was presented, and the process by which it generated natural language reports was described. A procedure by which configuration, in the form of graph reduction rules, could be used to interpret requirements information captured with the OMT Object Model modeled under Metaview was demonstrated.

The approach used by REVIEW is similar to, and in fact is based upon, that used in the GETS system [11]. However, REVIEW has an important difference from other systems that generate natural language descriptions from software specifications. Where other systems focus on a single specification environment or methodology, REVIEW is designed to interpret specifications in any of the large family of methodologies supported by Metaview.

Future research efforts include:

- evaluating and optimizing the graph generation and traversal algorithms;
- analysis of the types of reduction rules necessary for interpreting other requirements specification methodologies, including Structured Analysis, Structured Design, and OMT's Dynamic and Functional Models;
- investigating the production of reports at varying levels of detail.

Acknowledgements

The authors would like to thank NSERC and the Canadian National Centres of Excellence Program for their support of this research.

References

[1] Casty, A. Building Writing Skills: A Programmed Approach to Sentences and Paragraphs. Harcourt Brace Jovanovich, 1971.

[2] Chen, P. P. The Entity-Relationship Model: Toward a Unified View of Data. *ACM Transactions on Database*, vol. 1, no. 9, March 1976, pp. 9-36.

[3] Clifford, J. and Waterhouse, R. F. Sentence Combining: Shaping Ideas for Better Style. Bobbs-Merrill Educational Publishers, 1983.

[4] Dedourek, J. M., Sorenson, P. G., and Tremblay, J. P. Metasystems for Information Processing System Specification Environments. *INFOR*, vol. 27, no. 3, August 1989, pp. 311-337.

[5] The Metaview System. Technical Report TR96-13, Dept. of Computing Science, University of Alberta, 1994.

[6] Johnson, W. L., Feather, M. S., and Harris, D. R. Representation and Presentation of Requirements Knowledge. *IEEE Transactions on Software Engineering*, vol. 18, no. 10, October 1992, pp. 853-869.

[7] McAllister, A. J. Modeling Concepts for Specification Environments. Ph.D. thesis, Dept. of Computational Science, University of Saskatchewan, 1988.

[8] Mylopoulos, J., Borgida, A., Jarke, M., and Koubarakis, M. Telos: A language for Representing Knowledge About Information Systems. Dept. of Computer Science, University of Toronto, February 1990.

[9] Protsko, L. B., Sorenson, P. G., Tremblay, J. P., and Schaeffer, D. A. Towards the Automatic Generation of Software Diagrams. *IEEE Transactions on Software Engineering*, vol. 17, no. 1, January 1991, pp. 10-21.

[10] Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., and Lorensen, W. Object Oriented Modeling and Design. Prentice-Hall, 1991.

[11] Salek, A. K. GETS: A Natural Language Generator for Specifications. M.Sc. thesis, Dept. of Computing Science, University of Alberta, 1991.

[12] Salek, A. K., Sorenson, P. G., Tremblay, J. P., and Punshon, J. M. The REVIEW System: From Formal Specifications to Natural Language. *Proceedings of the First Int'l Conference on Requirements Engineering, Colorado Springs*, April 18-21, 1994, pp. 220-229.

[13] Sorenson, P. G., Tremblay, J. P., and McAllister, A. J. The Metaview System for Many Specification Environments. *IEEE Software*, vol. 5, no. 2, February 1988, pp. 30-38.

[14] Swartout, B. GIST English Generator. *AAAI-82 Proceedings of the National Conference on Artificial Intelligence*, August 1982, pp. 404-409.

[15] Zhuang, Y., Findeisen, P., and Sorenson, P. G. Object-Oriented Modeling in Metaview. Department of Computing Science, U. of Alberta, 1995.