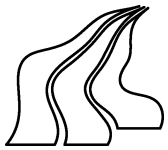
Aalborg University



Institute of Computer Science ♦ Fredrik Bajers Vej 7 ♦ 9220 Aalborg Øst.

TITLE:

The Aware Design Tool
A Tool Supporting a Data Warehouse Design Methodology

PROJECT PERIOD:

February 2nd. - May 26th. 2000

PROJECT GROUP:

E1-209a

TERM:

Dat 6

AUTHORS:

Carsten Nielsen
Flemming N. Larsen
Peter S. Kristiansen

SUPERVISOR:

Nectaria Tryfona

ABSTRACT:

This Masters Thesis describes the architecture of the Aware Design Tool. The Aware Design Tool supports a data warehouse design methodology. We briefly describe the advantages of utilizing a design methodology.

We describe the architecture of the Aware Design Tool. This includes a description of the communication among the components in the Aware Design Tool. Furthermore, we describe all the components in the Aware Design Tool.

We suggest expansions to the Aware Design Tool. This include a Controller, a Guardian and a conceptual query language.

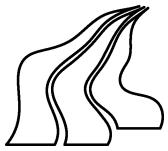
Finally, we conclude about our experiences regarding the Aware Design Tool as well as providing suggestions to future work.

NUMBER PRINTED: 8

NUMBER OF PAGES: 90

APPENDIX: 3

Aalborg Universitet
Institut for Datalogi ♦ Fredrik Bajers Vej 7 ♦ 9220 Aalborg Øst



TITEL:

The Aware Design Tool
A Tool Supporting a Data Warehouse Design Methodology

PROJEKT PERIODE:

2. Februar - 26. Maj 2000

PROJEKT GRUPPE:

E1-209a

SEMESTER:

Dat 6

FORFATTERE:

Carsten Nielsen
Flemming N. Larsen
Peter S. Kristiansen

VEJLEDER:

Nectaria Tryfona

SYNOPSIS:

Denne rapport beskriver arkitekturen af et data warehouse design værktøj, the Aware Design Tool. Programmet Aware Design Tool støtter en data warehouse design metodik. I denne rapport beskriver vi kort fordelene ved at bruge en design metodik.

Vi beskriver arkitekturen af værktøjet Aware Design Tool. Dette inkluderer en beskrivelse af kommunikationen mellem komponenterne i værktøjet Aware Design Tool.

Vi foreslår udvidelser til værktøjet Aware Design Tool. Disse foreslag inkluderer en Controller, en Guardian og et konceptuelt spørgesprog.

Til sidst konkluderer vi omkring vores erfaringer angående værktøjet Aware Design Tool, og vi giver foreslag til fremtidigt arbejde.

OPLAG: 8

SIDETAL: 90

APPENDIKS: 3

Preface

This Master's Thesis is the presentation of the results of group E1-209a's work at Aalborg University, Institute of Computer Science in spring 2000.

This report constitutes part I of the Master's Thesis. Part II of the Master's Thesis is a seperate report; The Aware Design Tool, A User Guide.

Literature references are written on the form [Ora99a]. A bibliography can be found on page 89. Figures are enumerated by chapter number followed by a consecutive number within the chapter. All references to elements in figures are written in italics.

Aalborg, May 26th. 2000

Peter S. Kristiansen

Carsten Nielsen

Flemming N. Larsen

Contents

1 Introduction

This chapter begins with a description of the motivation that has led to the creation of a data warehouse design tool, along with a presentation of the previous work that led to the architecture of the design tool. Following, our contribution to the data warehousing community in this report is described. At the end of this chapter, the contents of the rest of the report is described.

1.1 Motivation

This report focuse upon the creation of a data warehouse design tool. This design tool supports the data warehouse design methodology which was described in [NLK99]. Because of the theoretical advantages of utilizing this data warehouse design methodology, we decided to implement a tool to support the methodology.

The main advantage of utilizing a data warehouse design methodology is abstraction. That is, the designer is provided with the ability to design a data warehouse using high-level concepts. This has the advantage that at the conceptual design phase, the designer can work in conjunction with the end-users of the data warehouse, and thereby the resultant data warehouse schema should reflect the requirements of the end-user. At the logical design phase, the data warehouse designer is able to supply additional information, such as data types and attribute domains. The physical design phase represents the actual implementation of the data warehouse, and allows the designer to specify implementation issues, such as record size.

An important aspect of data warehouse modeling is the ability to limit the

data to be loaded into the data warehouse. That is, not all data from the source systems are wanted in the data warehouse. For example, data regarding customers below 18 years of age may not be of interest in the data warehouse. One way to handle this is to introduce explicit constraints as presented in [NLK99]. The constraints presented are:

- Existence Dependency constraints (EDs).
- Domain Existence Dependency constraints (DEDs).
- Equality Expressions (EEs).

In this report, the names of the above listed constraints have been altered in order to clarify the intension of these constraints.

Existence Dependency constraints (EDs) have been renamed to *Entity Constraints* (ECs). Entity constraints are used for constraining entity sets. That is, by specifying an entity constraint, the designer can limit instances of entities.

Domain Existence Dependency constraints (DEDs) have been renamed to *Attribute Constraints* (ACs). Attribute Constraints can be imposed on regular attributes[1], and are used to limit instances of attribute values.

Equality Expressions (EEs) have been renamed to *Summarizable Attribute Constraints* (SACs). These constraints are imposed on summarizable attributes. SACs are used for specifying how a summerizable attribute must be aggregated.

1.2 The AWARE DESIGN TOOL Prototype

A prototype of the AWARE DESIGN TOOL has been implemented. This section first provides a brief description of the requirements to a data warehouse design tool. Following, the choices regarding implementation issues of the prototype of the AWARE DESIGN TOOL are described.

In a previous report we identified and described a number of general requirements for a data warehouse design tool [NLK99]. These requirements, briefly sketched, are:

[1] A regular attribute is an attribute, that is not a summerizable attribute.

- The design phases of the data warehouse design methodology must be explicitly supported in order to take advantage of the benefits of this design methodology.

- It is desirable that the design tool is able to generate documentation based on the schemas. This would allow the data warehouse designer to verify the structure of the data warehouse regarding the data requirements provided by the end-users.

- The design tool should supply on-line help, in order to assist the designer of the data warehouse. The on-line help must include a description of the design methodology supported by the design tool, the data models and the languages supported in each design phase.

- The graphical user interface of the design tool must be intuitive and easy to use. This is necessary in order to reduce the time spent on learning how to use a design tool. Moreover, the intention of the design tool is that the designer should focus upon the design of the data warehouse, and not how to use the tool.

The prototype of the AWARE DESIGN TOOL supports the conceptual and logical design phase fully, while it does not support the physical design phase. At the conceptual design phase, the AWARE DESIGN TOOL supports the starER model, as described in [TBC99]. At the logical design phase, the AWARE DESIGN TOOL supports the Extended Snowflake Schema as described by [Kel99]. In addition, the AWARE DESIGN TOOL allows the data warehouse designer to specify explicit constraints at the conceptual design phse. The physical design phase is not supported in the AWARE DESIGN TOOL. Creation and management of the physical schema has been left for the Oracle 8.i DBMS.

For the implementation, Windows NT has been chosen as the platform for the prototype of the AWARE DESIGN TOOL. The actual implementation is not platform independent and therefore the AWARE DESIGN TOOL is not easily ported.

1.3 Our Contribution

This section contains a brief summary of the contributions provided by this report towards the data warehouse community.

The usefulness of utilizing a data warehouse design methodology already has been established, as can be seen in [NLK99]. Therefore, we propose that the architecture of a design tool should utilize a design methodology. In order to facilitate the communication between the various components of this design tool, we propose the utilization of metadata for this communication. More specifically, the starER Metadata Containers should be used to communicate conceptual schemas, and the Snowflake Metadata Containers should be used to communicate logical schemas.

We propose how to translate an Extended Snowflake schema into SQL *Plus statements. The translation enables the data warehouse schema, created in the AWARE DESIGN TOOL, to be implemented in an existing DBMS.

We propose how to translate the constraints specified on a starER schema into constraints for an Extended Snowflake schema. Furthermore, we provide means for translating constraints into SQL *Plus statements.

We propose the AWARE DESIGN TOOL being expanded by the inclusion of a Controller. This Controller should be able to dynamically maintain schemas. We propose inclusion of data security into a design tool. We argue why it is necessary to administrate user groups at the conceptual design phase, and we suggest how the AWARE DESIGN TOOL can facilitate such functionality.

Finally, we propose a data mining tool to extract and view data in a data warehouse. This data mining tool should be partly independent from the AWARE DESIGN TOOL. Moreover, we exemplify how a graphical query language could make it possible to specify conceptual queries.

1.4 Contents of the Report

In chapter 2, we present a methodology for data warehouse design. This methodology is based on the traditional database design methodology (see [BCN92] and [EN94]). We argue why a data warehouse design methodology should be utilized as well as describe the advantages gained by utilizing such a design methodology. Furthermore, the individual design phases of this methodology are described along with how the AWARE DESIGN TOOL supports data warehouse design methodology.

In chapter 3, the architecture of the AWARE DESIGN TOOL is described. First, the general architecture of the AWARE DESIGN TOOL is described. Following, the communication among the components in the AWARE DESIGN TOOL is described. Thereafter, the metadata containers used for communi-

cation within the tool are described. Then the parsers and generators used in the conceptual and logical design phases are described. After this, the translators used between each of the design phases supported in the AWARE DESIGN TOOL are described. The last section in chapter 3 describes the repository.

Chapter 4 describes the ideas for expanding the AWARE DESIGN TOOL. First a Controller is proposed. Subsequently, a brief description of the suggestions for how data security can be integrated into the AWARE DESIGN TOOL is presented. Finally, a conceptual query language is proposed. This query language should enable the end users to specify queries on a conceptual schema. For this purpose, we introduce a new tool, the *Aware Query Tool*.

Finally, chapter 5 provides conclusions upon our work as well as giving suggestions for future work.

2 Methodology

In this chapter, a data warehouse design methodology is described, along with the advantages of using a design methodology. This is followed by a description of the individual design phases in the design methodology. Finally, it is described how the Aware Design Tool supports the described design methodology.

2.1 A Data Warehouse Design Methodology

Databases are traditionally designed using a database design methodology that consists of three separate design phases (see [BCN92] and [EN94]). The database design is divided into the conceptual, logical, and physical design phases. The concept of having these three seperate design phases has been utilized toward the design of data warehouses by using a data warehouse design methodology [NLK99].

Figure 2.1 shows each of the design phases that exists in the data warehouse design methodology. Moreover, this figure shows the input and output of each of the design phases.

Designing a data warehouse starts with the data requirements supplied by the end-users of the data warehouse. These data requirements are processed in each of the design phases of the methodology. The result of the conceptual design phase is a conceptual schema that reflects the data requirements, based upon a specific conceptual data model. The conceptual schema is used as input for the logical design phase, which is based on a specific logical model, e.g. the relational data model.

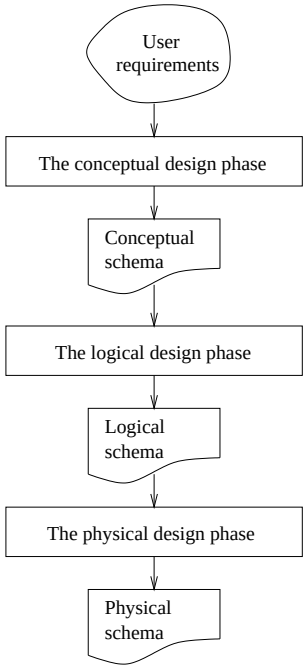


Figure 2.1: The phases of the data warehouse design methodology.

The logical design phase focuses on modelling logical aspects of the data requirements e.g., the structure of a schema. The result of the logical design phase, i.e. a logical schema, is used as input for the physical design phase. The physical design phase is the final design phase, which is dependent on a specific physical data model i.e., a specific DBMS system and a specific platform. In the physical design phase, a physical schema is created based on the logical schema. The physical schema constitutes the actual implementation of the designed data warehouse. The advantages of using a data warehouse design methodology are:

- Abstraction: A data warehouse design methodology allows placement of concepts at a proper level of abstraction, which reduces the complexity of the individual design phases. That is, high-level concepts such as entity sets, relationship sets, hierarchies etc. are modelled in the conceptual design phase, and concepts such as tables, data types, and attribute domains are modeled at the logical level. Low-level concepts that are dependent on a specific physical data model, such as block sizes and data partioning, are modelled in the physical design phase.
- Documentation: The output of each design phase i.e., a schema, can

be used as part of the data warehouse documentation.

- Evaluation: The schemas between the various design phases can be used for evaluating the correctness of the designed schema. This evaluation should be performed by the persons that are involved in the design phases.

- Maintenance: The data warehouse becomes easier to maintain by following a design methodology. A change upon the data warehouse will be made in a specific design phase. If the change involves conceptual aspects, the change will be made in the conceptual design phase. Similarly, if the change involves either logical or physical aspects, the change will be be made at the logical or physical design phase respectively. Changes made in the conceptual or logical design phases are propagated to the underlying design phases.

- Reusability and Portability: The schemas from the conceptual and logical design phases can be reused, which is useful if the data warehouse must be implemented on different platforms. A conceptual schema is reusable, due to the fact that a conceptual schema must be independent of any implementation related issues. A logical schema is independent of the actual physical implementation, but reusable for the same logical data model only. That is, if a logical schema is based on e.g., the relational data model, the logical schema is reusable another DBMS that supports the relational model.

Now that the advantages of using a data warehouse design methodology have been described, each of the individual design phases of the methodology are described in the following.

2.2 The Design Phases

In this section, the characteristics and advantages of each design phase in the data warehouse methodology are described. First the conceptual design phase is described. Secondly, the logical design phase is described, and finally the physical design phase is described.

2.2.1 The Conceptual Design Phase

The purpose of the conceptual design phase is to model a data warehouse based on the data requirements provided by the end-users. The conceptual design phase focuses on modelling concepts by abstraction. That is, the conceptual design phase focuses on modelling concepts that are well-known to the end-users. Thus, the conceptual design phase should always strive to utilize as simple and expressive a conceptual data model as possible. As the notions of simplicity and expressiveness are conflicting, a balance between these two terms must be reached.

The conceptual data model used in the conceptual design phase must be independent of any implementation issues. This allows the design of the data warehouse to initiate before decisions have been made regarding what technical platform to use for implementing the data warehouse.

2.2.2 The Logical Design Phase

The purpose of the logical design phase is to construct a logical schema based on the conceptual schema, provided from the conceptual design phase. The logical schema is required in order to map the concepts defined in the conceptual schema into concepts that exists in a specific logical data model, e.g., the relational data model. This implies that a specific logical data model have been chosen regarding the implementation of the data warehouse. Moreover, a specific DBMS *type* have chosen, which is based on this logical data model. That is, if the relational data model has been chosen as the logical data model, the data warehouse will be implemented on a relational DBMS.

The logical design phase is independent of the choice of a specific DBMS. This means that the logical schema can be reused in different DBMS' that supports the same logical data model.

At the logical design phase, additional information must be provided on the logical schema, such as defining data types and domains on attributes. Furthermore, at the logical design phase, it is possible to restructure the logical schema. For example, it can be necessary to restructure the logical schema in order to meet the end-users requirements regarding fast query performance.

2.2.3 The Physical Design Phase

At the physical design phase, the logical schema created in the logical design phase is mapped into an actual implementation of the data warehouse. That is, in this design phase the physical schema is created. Thus, the physical schema is completely dependent on the specific DBMS system chosen for the actual implementation of the data warehouse.

At the physical design phase, it is possible to tweak implementation issues. This means that the designers of the data warehouse are able to modify details in the physical schema if necessary, such as changing the record size, index structure, data partioning etc. of the underlying database.

In the following section, the support of the design methodology into the AWARE DESIGN TOOL is described along with the basic concepts of the Aware Design Tool.

2.3 The AWARE DESIGN TOOL Supporting the Methodology

The design methodology described above provides a plethora of advantages. Because of these advantages, we have decided to support the methodology in the AWARE DESIGN TOOL.

The AWARE DESIGN TOOL is used for designing conceptual and logical schemas, but not physical schemas. Thus, the AWARE DESIGN TOOL only supports the conceptual and logical design phases. The physical design phase has been omitted from the AWARE DESIGN TOOL as the main focus of this tool have been to support the conceptual and logical design phase.

The process of designing a data warehouse by utilizing the AWARE DESIGN TOOL is described in the following:

1) Creation of the conceptual schema. The conceptual schema is designed in order to model the concepts, which the end-users wants to be modelled in the data warehouse.

2) Transformation of the conceptual schema into a logical schema. When the conceptual schema has been designed, it is translated into a corresponding logical schema.

3) Modification of the logical schema. After translating the conceptual schema into a logical schema, the designer has the opportunity to modify the logical schema. Modifications on the logical schema include restructuring the schema, specifying data types and the domain of attributes.

4) Translation of the logical schema into SQL statements. When the modifications of the logical schema have been made, the logical schema is translated into SQL statements. These SQL statements are used as input to the physical design phase.

5) Excecution of the SQL statements in the DBMS. The physical schema is created by executing the SQL statements provided from the logical design phase. Note that the AWARE DESIGN TOOL does not support the physical design phase.

Figure 2.2 shows the design phases of the AWARE DESIGN TOOL. The specific activities that takes place in each of these design phases are also shown on this figure.

The data model supported in the AWARE DESIGN TOOL for the conceptual design phase is the starER model as presented by [TBC99], and enriched with constraints by [NLK99].

The conceptual design phase is the first phase the designer encounters in the AWARE DESIGN TOOL. It is assumed that at this point, the designer has acquired knowledge about the data requirements provided by the end-users. A conceptual schema is designed either by drawing a starER schema, by writing StarLanguage source code, or by a combination of these two design approaches. For more information about the StarLanguage, see appendix A. Note that the two representations of a starER schema are interchangeably. Thus, in the AWARE DESIGN TOOL it is possible to transform a graphical representation of a starER schema into StarLanguage source code and vice versa (see figure 2.2).

In order to transform a graphical representation of the starER schema into StarLanguage source code, the graphical starER schema is first converted into an instance of the starER Metadata Containers (see figure 2.2. The starER Metadata Containers are described in section 3.3.1. Secondly, this instance is converted into StarLanguage source code, which completes the process of the conversion. If StarLanguage source code must be converted into a graphical representation of a starER schema, this process is reversed.

A logical schema is constructed by translating the conceptual schema i.e., a starER schema, into an Extended Snowflake schema. In the AWARE DESIGN

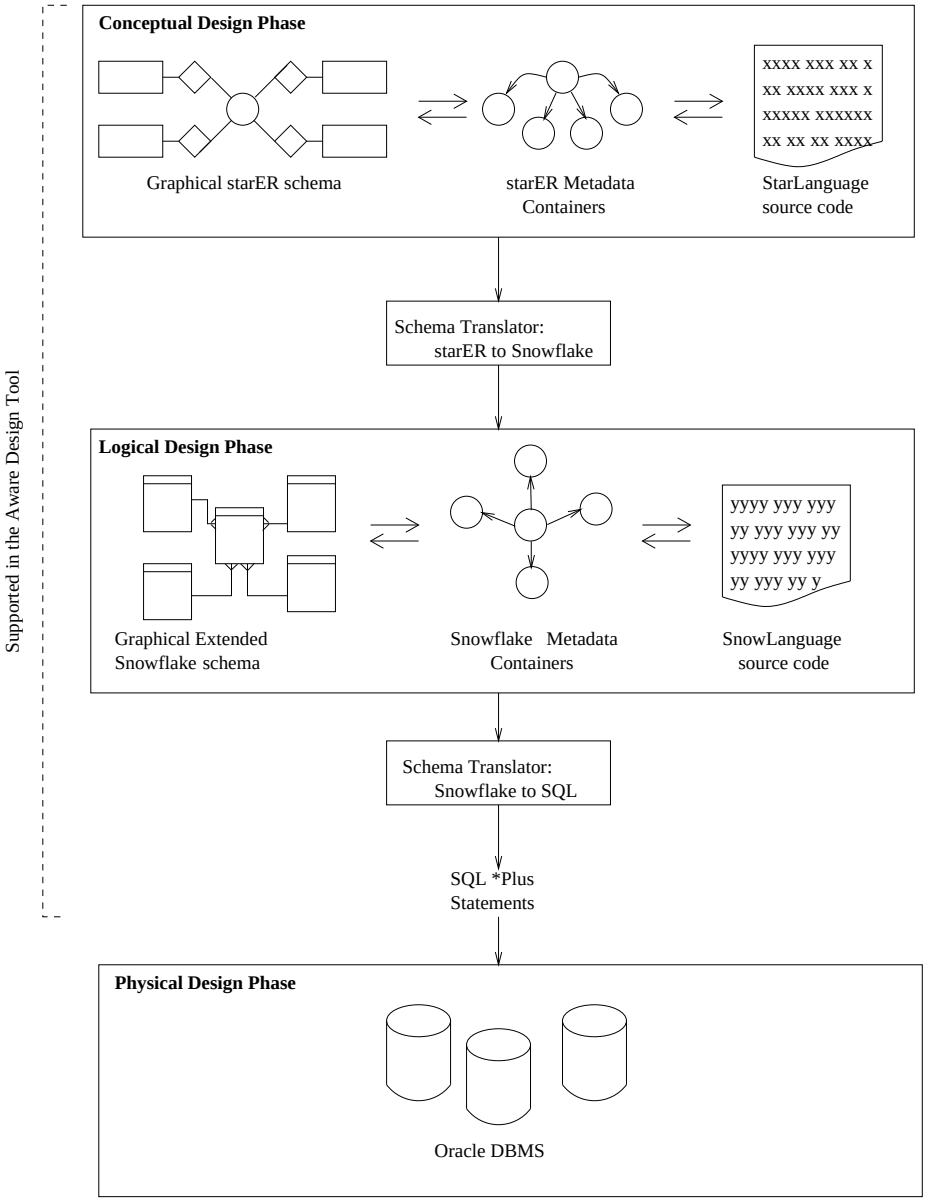


Figure 2.2: The design phases supported in the AWARE DESIGN TOOL.

TOOL this is done automatically by a built-in schema translator (see figure 2.2. When the schema translation have been made, it is possible for the designer to modify the logical schema.

An Extended Snowflake schema can be modified either graphically, by writting SnowLanguage source code, or by a combination of these two design procedures. For more information about the SnowLanguage, see appendix B, which provides the syntax of this language. Note that it is possible to transform an Extended Snowflake schema into SnowLanguage source code and vice versa in the AWARE DESIGN TOOL (see figure 2.2.

In order to transform a graphical representation of the Extended Snowflake schema into SnowLanguage source code, the graphical schema is first converted into an instance the Snowflake Metadata Containers. The Snowflake Metadata Containers are described in section 3.3.2. Secondly, this instance is translated into SnowLanguage source code, which completes the process of converting the graphical Extended Snowflake schema. If SnowLanguage source code is to be converted into a graphical representation of an Extended Snowflake schema, this process is reversed.

When the logical schema has been designed i.e., when the logical schema fulfills the requirements, the logical schema is translated into Oracle SQL *Plus statements [Ora99a]. These SQL *Plus statements are used as input for the physical design phase. For the physical design phase, the Oracle 8i DBMS have been chosen. The SQL *Plus statements must be executed in the physical design phase in order to generate the physical schema of the data warehouse.

As mentioned earlier, the physical design phase is not supported by the AWARE DESIGN TOOL. This is due to the fact that this design phase deals with the actual implementation of the data warehouse using a specific database supplied by a specific DBMS system. This task is considered to be out of the scope of the AWARE DESIGN TOOL.

3 Architecture of the Aware Design Tool

In this chapter, the architecture of the AWARE DESIGN TOOL is described. Firstly an overview of the general architecture AWARE DESIGN TOOL is provided. Then the communication among the components in the AWARE DESIGN TOOL is described. Furthermore, the functionality and internal structure of the components in the architecture will be described in greater detail.

3.1 The General Architecture

The architecture of the Aware Design Tool is divided into five components, the *Graphical User Interface* (GUI), the *Repository*, the *Source code parsers*, the *Source code generators* and the *Schema translators* as can be seen in figure 3.1.

The GUI component contains the *Visualizer*, which is a sub-component responsible for visualizing the conceptual and logical schemas. Moreover, the GUI contains the *Method knowledge* sub-component which provides on-line help. Furthermore, the GUI contains the *Method object schema* sub-component, which contains the definitions of the supported models of the

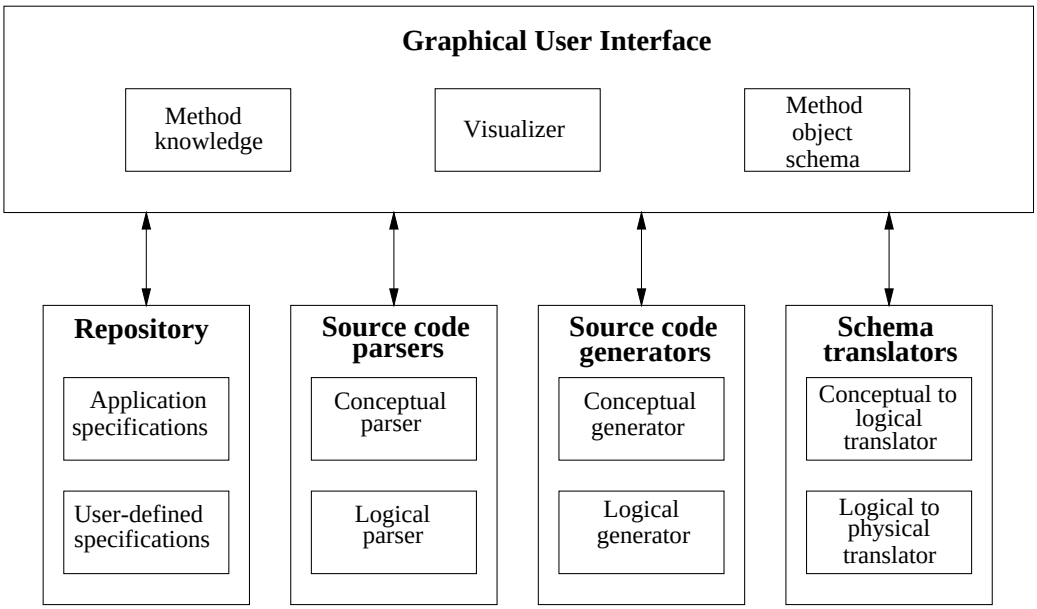


Figure 3.1: Architecture of the AWARE DESIGN TOOL.

design tool.

The *Source code generators* component handles the conversion of schemas into source code. This component consists of two sub-components, the *Conceptual generator* and the *Logical generator*. The *Conceptual generator* converts starER schemas into StarLanguage source code, and the *Logical generator* converts Extended Snowflake schemas into SnowLanguage source code.

The *Source code parsers* component are used for converting source code into schemas. The *Source code parsers* contains the sub-components *Conceptual parser* and *Logical parser*. The *Conceptual parser* enables the AWARE DESIGN TOOL to transform StarLanguage source code into a starER schema. At the logical design phase, the transformation of SnowLanguage into an Extended Snowflake schema is handled by the *Logical parser*.

The *Schema translators* component is used for translation schemas between the different design phases. The *Schema translators* contains the *Conceptual to logical translator* and the *Logical to physical translator*. The *Conceptual to logical translator* sub-component handles translation of starER schemas into Extended Snowflake schemas. The sub-component *Logical to physical translator* translates Extended Snowflake schemas into SQL *Plus statements.

The last component of the AWARE DESIGN TOOL is the *Repository*. This

component is responsible for storing schemas from the conceptual and logical design phases. The repository consists of two sub-components: the *Application specifications*, and *User-defined specifications*. The *Application specifications* is the sub-component of the *Repository* that stores and loads the schemas created at the various design phases of the AWARE DESIGN TOOL. The *User-defined specifications* is the part of the *Repository* that enables the AWARE DESIGN TOOL to store and retrieve user-defined components.

3.2 Communication among the components in the AWARE DESIGN TOOL

In this section, the communication among the various components in the architecture of the AWARE DESIGN TOOL is briefly described.

The GUI component is responsible for controlling all the other components in the AWARE DESIGN TOOL. In addition, none of the other components communicate directly with each other. The benefit of this architecture is that it is easy to replace, modify, or test the components that the GUI component is communicating with. Each of the components that the GUI is communicating with can be replaced, modified, or tested independently, as these components are not dependent upon each other.

Figure 3.2 provides an overview of the general communication among the components in the architecture of the AWARE DESIGN TOOL.

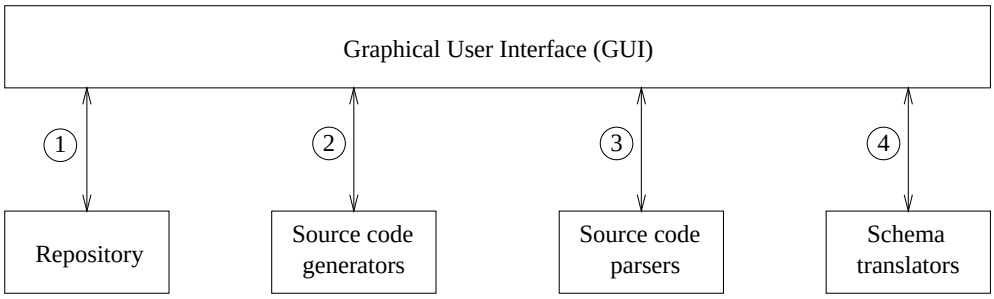


Figure 3.2: The general communication among the components.

In the following, the general communication among the GUI component and each of the other components is described.

1. The *Repository* provides functionality for storing and loading conceptual and logical schemas. The GUI requests the *Repository* to store a new or modified schema, or to load a previously stored schema.

2. When source code view is required by the desinger, the GUI sends the schema to the *Source Code Generator*. The *Source Code Generator* then generates source code based on the schema components, structure and constraints, and returns the source code to the GUI. The GUI then display the source code.

3. In order to view a schema based on the source code of the schema, the GUI sends the source code to the Source Code Parser. The Source Code Parser constructs the schema by parsing the source code, and sends the constructed schema to the GUI. The GUI then displays the graphically schema.

4. *Schema translators* are used for translating a schema from one design phase to a schema in another design phase. That is, a schema translator is used for translating a conceptual schema into a logical schema, and a schema translator is used for translating a logical schema into a physical schema.

3.3 Metadata Containers

In this section the metadata containers for the AWARE DESIGN TOOL are described. The metadata containers are used for communication among the various components in the AWARE DESIGN TOOL.

The metadata containers are object-oriented and consist of two separate class hierarchies. One class hierarchy is provided for communication at the conceptual design phase, and the other class hierarchy is provided for communication at the logical design phase.

In this context, we provide a definition of metadata:

Metadata: *Data about a schema.*

The metadata containers for the conceptual design phases are used for communicating starER schemas only, and the metadata containers for the logical design phase are used for communicating Extended Snowflake schemas only.

The data about a schema includes all the elements that exists in the schema e.g., entity sets, relationship sets, constraint definitions etc. This includes the structure of the schema i.e., how the elements are connected and graphical information needed to display the elements in a schema.

In the following two subsections the metadata containers used for communication among the components are described.

3.3.1 starER Metadata Containers

In this subsection the metadata containers used at the conceptual design phase are described. These metadata containers are called the *starER Metadata Containers*. Figure 3.3 shows the class hierarchy of the starER Metadata Containers using the object-oriented Unified Modelling Language (UML) [MMMNS97]. In the following, the classes in this hierarchy are described. Throughout the remainder of this section the term component is used. This does not refer to any of the components of the AWARE DESIGN TOOL, but an abstract notion of the parts of the metadata containers.

Schema

The *Schema* class is used for representing a starER schema. The *Schema* class is the main class in the class hierarchy of the starER Metadata Containers. At the conceptual design phase, the various components in the AWARE

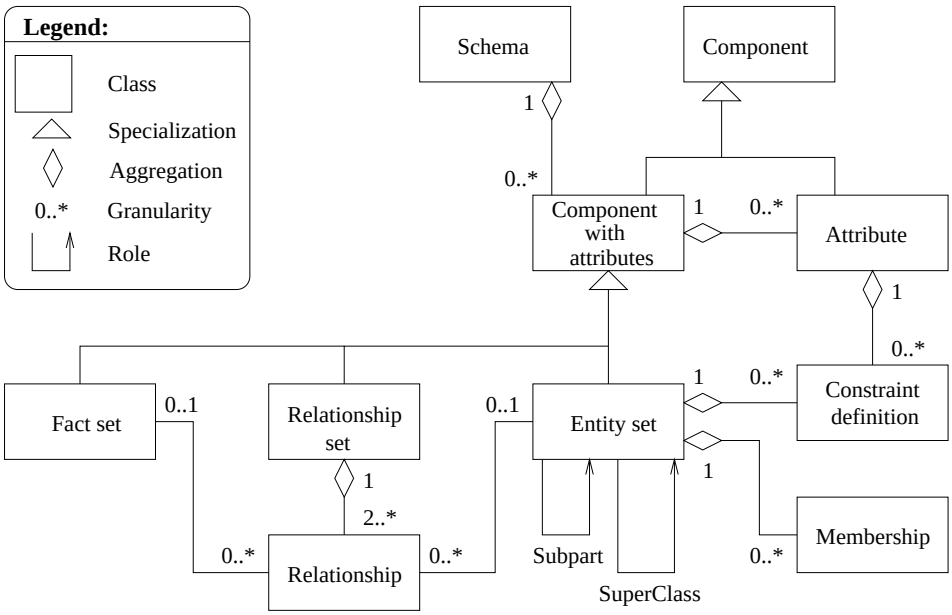


Figure 3.3: The starER metadata containers.

DESIGN TOOL communicates by sending and recieving an instance of the *Schema* class. An *Schema* instance contain zero to many instances of the *Component with Attribute* class. These instances are the main components in a schema.

Component

All components in a starER schema are specializations of the *Component* class. The *Component* class contains data about the ID and graphical position of components in a schema. The ID of a component must be unique. The graphical position of a *Component* is used when the component is visualized.

Component with Attributes

The *Component with Attributes* is a specialized *Component* class. This class is used for representing components that has attributes. That is, the *Component with Attributes* class is a generalization over the classes *Fact Set*, *Entity Set*, and *Relationship Set*. Instances of the *Component with Attributes* class can contain zero to many attributes.

Attribute

The *Attribute* class is a specialized *Component*. This class is used for representing attributes on fact sets, entity sets, or relationships sets in a schema. The *Attribute* class contains data about the type of the attribute, which can be one of the following: a regular attribute, a key or a summarizable attribute as type flow, stock, or value-per-unit.

Constraint Definition

In order to model explicit constraints in the starER model, the *Constaint Definition* class is used. The *Constraint Definition* class is used for modelling constraints on any component that can have constaints. A *Constraint Definition* consists of a text string containing StarLanguage source code.

Currently, it is possible to specify Attribute Constaints (ACs), Summarizable Attribute Constraints (SACs), and Entity Constraints (ECs) in a starER schema. Therefore only instances of the *Attribute* class and the *Entity* class can contain constraint definitions.

In order to model ACs and SACs on an attribute, an *Attribute* instance can contain zero to many constraint definitions. Moreover, an *Entity Set* instance can contain zero to many constraint definitions in order to model ECs on an entity set. Note, that both entity sets and attributes can contain multiple constraint definitions.

Entity Set

The *Entity Set* class is mainly used to model entity sets. This class is also used for modelling aggregated and specialized entity sets.

In order to model aggregated and specialized entity sets, a single *Entity Set* instance can reference zero to many other *Entity Set* instances. An *Entity Set* instance that references another *Entity Set* instance has a role as either being a subpart of an aggregated entity set or being a super class of a specialized entity set.

Membership

The *Membership* class is used to model hierarchies. A membership between two entity sets is modelled on the entity set of lower granularity in the hierarchy. That is, the *Entity Set* instance of the lower granularity must contain a *Membership* instance, and this *Membership* instance must be reference the *Entity Set* instance of the higher granularity. Note that an entity set can

take part in zero to many memberships. Thus, an *Entity Set* instance can contain zero to many *Membership* instances.

A *Membership* instance contains data about the cardinality of the membership. The cardinality of a *Membership* can be either strict, complete, or non-complete.

Relationship Set and Relationship
The *Relationship Set* class is used for modelling binary and high-order relationship sets. In order to model a relationship in a relationship set, the *Relationship* class is used. A *Relationship* instance references either a *Fact Set* or an *Entity Set* instance.

A *Relationship Set* instance contains the number of *Relationship* instances that corresponds to the order of the relationship set. Hence, in order to model a binary relationship set, a *Relationship Set* instance must contain two *Relationship* instances. Note that a *Relationship Set* instance must contain at least two *Relationship* instances. That is, a relationship set cannot be unary.

A *Relationship* instance contains data about the granularity of the related entity set or fact set, which can be either *one* or *many*.

This concludes the description of the starER Metadata Containers. The following subsection will continue with a description of the Snowflake Metadata Containers.

3.3.2 Snowflake Metadata Containers

In this subsection the metadata containers used at the logical design phase are described. These metadata containers are called the *Snowflake Metadata Containers*.

The *Snowflake Metadata Containers* are used for communicating Extended Snowflake schemas. Although these containers are used for communicating Extended Snowflake schemas, they can also be used for communicating Star and Snowflake schemas. This is possible because these schemas uses the same components i.e., fact tables and dimension tables. The only difference among these schemas is the allowed number of fact tables[1], and the possibility of

[1] A Star schema and a Snowflake schema contain a single fact table only, whereas Extended Snowflake schemas can contain several fact tables.

having hierarchical structures[2].

Figure 3.4 shows a class hierarchy of the Snowflake Metadata Containers. In the following, the various classes in this hierarchy are described.

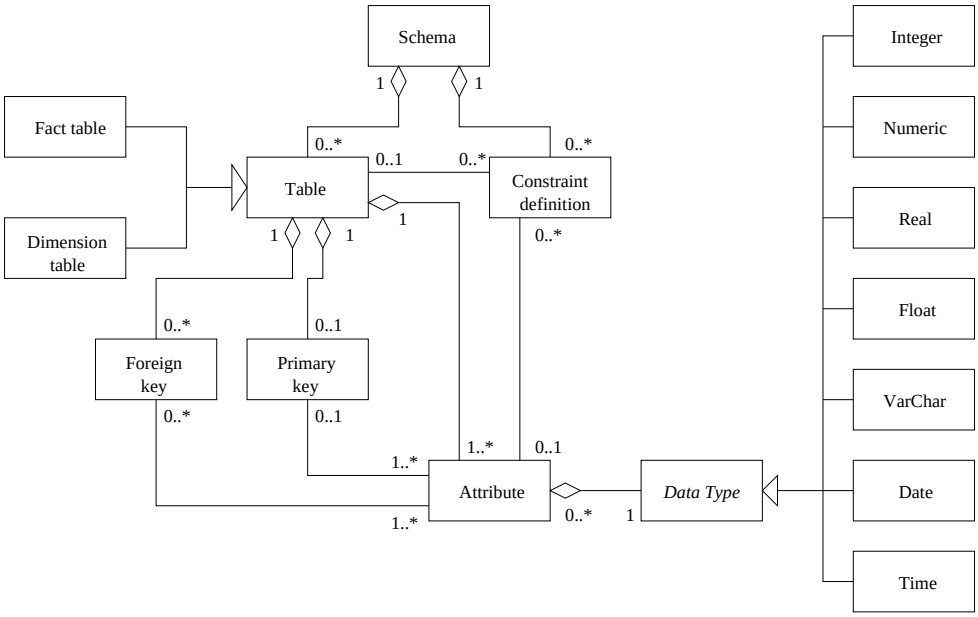


Figure 3.4: The Snowflake metadata containers.

Schema

The *Schema* class is used for representing Extended Snowflake schemas. The *Schema* class is the main class in the class hierarchy of the Snowflake Metadata Containers. At the logical design phase, the components in the AWARE DESIGN TOOL communicates by sending and receiving an instance of the *Schema* class.

A *Schema* instance contain zero to many instances of the *Table* class. These *Table* instances are the main components in the schema. In addition, a *Schema* instance contain zero to many *Constraint Definition* instances.

Table, Fact Table, and Dimension Table

The *Table* class is used for representing tables in a Snowflake schema. The *Table* class is a generalization over the specialized classes *Fact Table* and

[2]Hierarchical structures are allowed on Snowflake schemas and Extended Snowflake schemas only.

Dimension Table. The specialized versions of the *Table* class are used to distinguish the semantics of a table. That is, the *Fact Table* class is used for representing fact tables and the *Dimension Table* is used for representing dimension tables in the Extended Snowflake schema.

Only fact tables can contain facts [Kel99]. If no primary key is defined for a fact table explicitly, the primary key is constituted of all the foreign keys in the fact table.

Dimension tables contains descriptive data about facts [Kel99]. In addition, a primary key must be specified on a dimension table, as a dimension table is always refered to by another dimension or fact table.

The *Table* class contains data about the ID of the table and the graphical position of the table in the schema. The ID of a table must be unique in order to distinguish the tables in a schema. The graphical position of a table is used when the table is visualized.

Attribute
The *Attribute* class is used to represent an attribute on a table. A *Table* instance must contain at least one *Attribute* instance [3].

It is possible to specify a *not null* constraint for an *Attribute* instance. If a *not null* constraint is specified for an *Attribute* instance, this means that no instance of the attribute is allowed to contain the *null* value. This is useful when an attribute is part of a primary key, as primary keys are not allowed to be null [SKS97].

Data Type
At the logical phase, a data type must be specified for each attribute. This implies that an *Attribute* instance must contain an instance of a *Data Type* class.

The *Data Type* class is an abstract class of the data type classes: *Integer*, *Numeric*, *Real*, *Float*, *VarChar*, *Date*, and *Time*. These classes are used for specifying the data type of an *Attribute* instance, including the domain, precision, format etc.

Integer
The *Integer* class is used for representing signed integers of any length. An *Integer* class contains data about minimum and maximum values, if specified.

[3] A table that does not contain any attributes cannot contain any data. Thus, an empty table on a schema can be left out of the schema.

Numeric
In order to represent fixed-sized and signed numbers, the *Numeric* class is used. This class contains data about the number of decimals and the precision of a number, as well as data about minimum and maximum values, if specified.

Real
The *Real* class is used for representing real numbers of any length. This class contain data about the precision, and number of decimals, as well as data about minimum and maximum values, if specified.

Float
In order to represent fixed-sized floating-point numbers, the *Float* class is used. This class contains data about the size of a floating point number, as well as data about minimum and maximum values, if specified.

VarChar
The *VarChar* class is used for representing variable-sized strings. This class contains data about the upper limit of the number of characters that the string can contain.

Date
In order to represent dates, the *Date* class is used, as well as data about minimum and maximum values, if specified.

Time
The *Time* class is used for representing time instants, as well as data about minimum and maximum values, if specified.

Primary Key
An *Attribute* instance does not contain any data whether it is a primary key or not. In order to specify that an *Attribute* instance is a part of a primary key, the *Primary Key* class must be used.

A *Primary Key* instance is included in a table, and contains data about which attributes that takes part of the primary key of the table. Note that a *Table* instance is allowed to contain a single *Primary Key* instance only.

Foreign Key
In order to specify that an *Attribute* instance is part of a foreign key, the *Foreign Key* class must be used. It is possible to include zero to many foreign keys instances in a table. Each of these instances contain data about which attributes that are part of the foreign key.

Constraint Definition

In order to model constaints in the Extended Snowflake model, the *Constaint Definition* class is used. The *Constaint Definition* class is a general class used for modelling constraints on any table or attribute that has a constraint i.e., a *Table* or an *Attribute* instance. A *Constraint Definition* consists of a text string containing SnowLanguage source code.

3.4 Parsers and Generators

A schema in the AWARE DESIGN TOOL can be designed either graphically, by writing source code, i.e., a textual description of a schema using a language or a combination of these. This benefits the data warehouse designer who can choose between drawing the schema graphically and using a language by writing source code, which describe the contents of the schema.

The main advantage of drawing the schema is that the data warehouse designer is able to have an intuitive view of the schema. The main advantage of writing source code for the schema using a language is that the source code is in a textual form, which can be used for transferring the schema to another design tool, another platform, and for storing the schema. Moreover, a designer skilled in writing source code will be faster in designing schemas by writing than by drawing.

In order to combine the advantages of the two different approaches to designing a schema, the design tool must make it possible to transform a graphical schema into source code and vice versa. However, in order to make this possible, the language for describing schemas and the graphical representation of the schema must be equivalent. That is, the two approaches must be equal in expressiveness and power.

In this context, we provide two definitions for translating a schema into source code and vice versa.

Parser: *A component, that takes source code as input and produces a graphical schema.*

The output of the parser is metadata containers, which contains the produced schema in terms of metadata. This can be seen in figure 3.5.

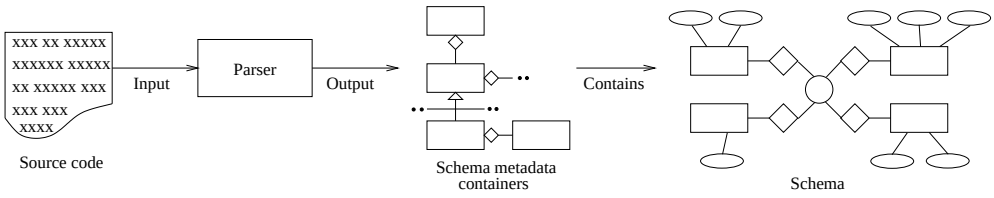


Figure 3.5: The input and output of a parser.

Generator: *A component, that takes a graphical schema as input, and generates source code as output.*

The input and output of a generator can be seen in figure 3.6.

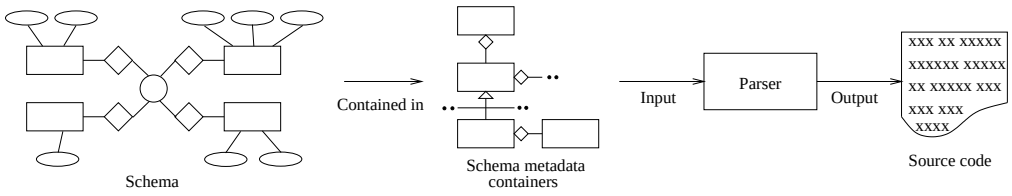


Figure 3.6: Input and output of a generator.

Hence, in order to provide the functionality for translating a graphical representation of a schema into source code *and* the other way around, both a generator and a parser is needed.

The AWARE DESIGN TOOL supports a design methodology allowing the designer to specify schemas at a conceptual phase, and modify schemas at a logical phase. Hence, the design tool must provide a parser and a generator at both the conceptual and the logical phase, in order to allow the designer to choose whether to design the schemas graphically or write source code for the schemas. In the following subsections, the parsers and generators used in the conceptual and logical design phases are described.

3.4.1 StarLanguage Parser & Generator

At the conceptual design phase, starER schemas are designed graphically by drawing starER schemas or by textually writing StarLanguage source code (the StarLanguage syntax is defined in appendix A).

Figure 3.7 shows the parser and generator provided in the AWARE DESIGN TOOL for the conceptual design phase. The *StarLanguage Generator* takes a starER schema contained in the *starER Metadata Containers* (described in 3.3.1) as input, and produces StarLanguage source code as output. The StarLanguage Parser takes StarLanguage source code as input, and produces a starER schema contained in the starER Metadata Containers as output.

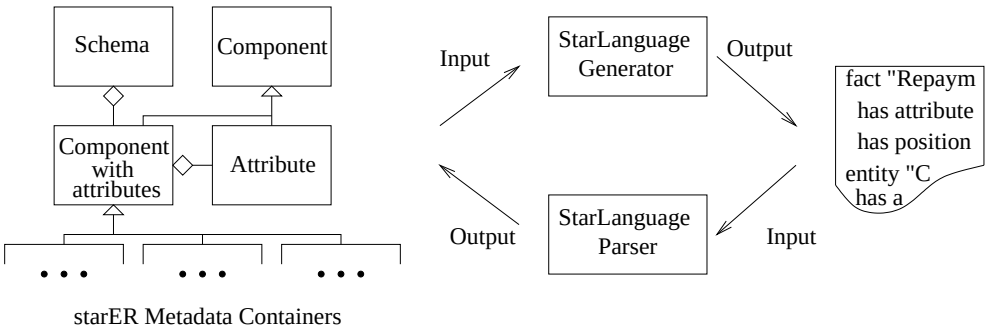


Figure 3.7: Input and output of the StarLanguage Parser and Generator.

3.4.2 SnowLanguage Parser & Generator

At the logical design phase, Extended Snowflake schemas are designed graphically by drawing Extended Snowflake schemas or, textually by writing SnowLanguage source code or a combination (the SnowLanguage syntax is defined in appendix B).

Figure 3.8 shows the parser and generator that are provided in the AWARE DESIGN TOOL for the logical design phase. The *SnowLanguage Generator* takes an Extended Snowflake schema contained in the Snowflake Metadata Containers (described in 3.3.2) as input, and produces SnowLanguage source code as output. The SnowLanguage Parser takes SnowLanguage source code as input, and producs a Extended Snowflake schema contained in the Snowflake Metadata Containers as output.

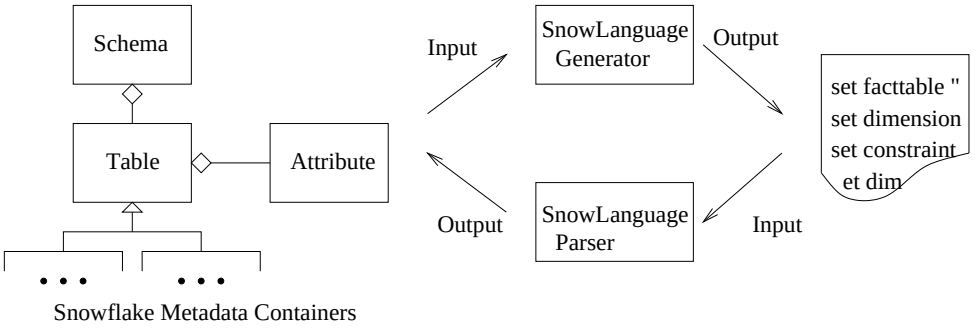


Figure 3.8: Input and output of the SnowLanguage Parser and Generator.

3.5 Schema Translators

When a starER schema has been designed at the conceptual design phase, it is translated into an Extended Snowflake schema. At the logical design phase, it is possible to specify data types and domains on attributes. Also, it is possible to restructure Extended Snowflake schemas. When the logical phase is finished i.e., when the Extended Snowflake schema reflect all required changes, the Extended Snowflake schema must be translated into SQL statements. These statements are used for creating the physical schema of the data warehouse and implement the constraints specified for the data warehouse.

The AWARE DESIGN TOOL is made to assist the designer with designing conceptual schemas, translating a conceptual schema into a logical schema, restructuring the logical schema, and translating a logical schema into a physical schema. In the AWARE DESIGN TOOL, two of these processes are achieved automatically. That is, the AWARE DESIGN TOOL is able to translate a starER schema into an Extended Snowflake schema, and is able to translate an Extended Snowflake schema into SQL *Plus statements utomatically. However, a requirement is that a schema,which is about to be translated, must be a valid schema.

3.5.1 starER to Extended Snowflake

The schema translator between the conceptual and logical design phase takes a starER schema expressed by the starER Metadata Containers as input, and produces an Extended Snowflake schema expressed by the Snowflake Metadata Containers as output. This is illustrated in figure 3.9.

The translator uses the translation rules provided in appendix C for translating a starER schema into an Extended Snowflake schema. The translation rules provide a proper translation of the elements and constraints in starER schemas into dimension tables, fact tables, and logical constraints for the Extended Snowflake schema.

No data type is defined on an attribute at the conceptual design phase, as the conceptual design phase is focused on modelling concepts of a data warehouse. However, at the logical design phase, a data type must be defined on attributes. If no data type is specified on an attribute in the logical design phase, a default data type is provided for the attribute [4]. Thus, as no data

[4] In the AWARE DESIGN TOOL, the default data type is a *varchar(0)*.

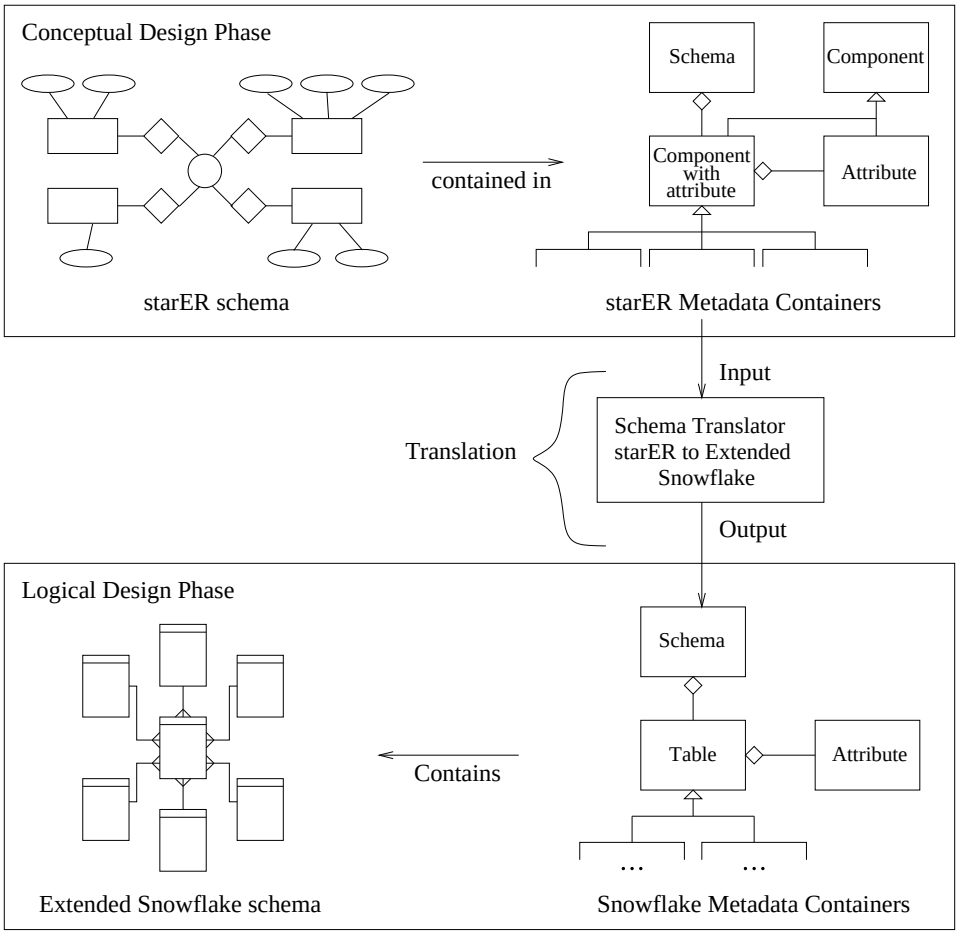


Figure 3.9: The translator between the conceptual and logical design phase.

type is specified on a conceptual schema, the schema translator used between the conceptual and logical design phase provides the default data type on the translated attribute in the resulting logical schema.

In the following, we continue by describing the schema translator provided in the AWARE DESIGN TOOL between the logical and the physical design phase.

3.5.2 Extended Snowflake to SQL

The schema translator between the logical and physical design phases takes an Extended Snowflake schema expressed by Snowflake Metadata Containers as input, and produces a set of files containing SQL *Plus statements as output. This is illustrated in figure 3.10.

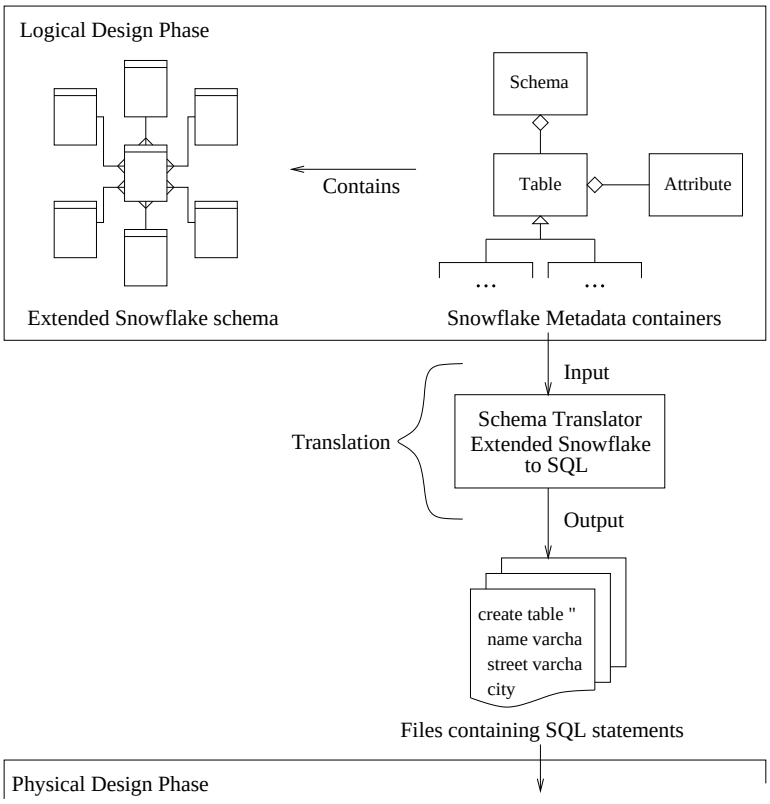


Figure 3.10: The translator between the logical and physical design phase.

The schema translator produces several files containing SQL statements us-

ing the SQL Data Definition Language (DDL) and the SQL Data Manipulation Language (DML). One of these files contains DDL statements only, which are used for defining and creating all the tables that must exists in the physical schema. In addition, the DDL statements are used for defining *integrity* constraints on the various tables. The integrity constraints guard against accidential damage to the data warehouse [SKS97], and are used for constraining the domain of attributes.

The other produced files contains DML statements, which are used for manipulation of the data in the data warehouse. Note that the facts and descriptive data about facts must be preserved i.e., this data is read-only [Mat96]. The produced DML statements are used only for the purpose of aggregating existing data in the data warehouse. That is, the aggregated data[5] does not affect existing data in the data warehouse, but is provided for the data warehouse as additional data.

Translation of a Logical Schema to SQL
When the schema translator translates an Extended Snowflake schema into SQL statements, each table in the Extended Snowflake schema is translated into a CREATE TABLE statement [Ora99b], which specifies:

- the table name,
- the name and data type of each attribute that exists on the table,
- the primary key of the table,
- foreign keys of the table, and
- integrity constraints on the attributes of the table.

If one or more integrity constraints are defined on attributes of a specific table, the CHECK clause [Ora99b] is used in conjunction with the CREATE TABLE statement in order to preserve the integrity constraints. The CHECK clause specifies a condition that must be checked for each row in the table, when new data is inserted into table or when existing data in the table is modified [SKS97]. New data can only be inserted into the table if the condition of the CHECK clause is not violated. Moreover, data can only be inserted into the data warehouse if the condition of the CHECK clause is not violated.

[5]The aggregated data is data of coarser granularity, which can be derived from the existing data in the data warehouse.

In the following, we provide an example of how a table is translated into a SQL statement. Figure 3.11 shows the dimension table *Inventory*, which contains the primary key *inventory*, the foreign key *store*, and the attribute *value*. An integrity constrain is defined on *value*, which specifies that the attribute value must be positive.

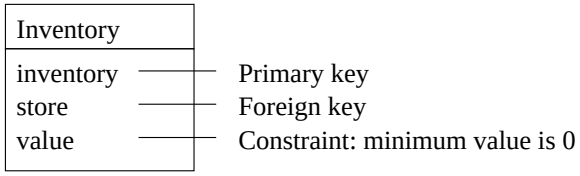


Figure 3.11: The Inventory dimension table.

This dimension table is translated into the following SQL statement:

```
CREATE TABLE Inventory
   (inventory VARCHAR(30),
    store VARCHAR(25),
    value REAL,
    PRIMARY KEY (inventory),
    FOREIGN KEY (store) REFERENCES Store,
    CHECK (value >= 0))
```

The domains of the *inventory*, *store*, and *value* attributes reflect the domains specified for these attributes on the *Inventory* table at the logical design phase. By default, an attribute can assume null values. An attributes which takes part in a primary key is not allowed to assume a null value [SKS97]. Note that the foreign key of the *Inventory* table refers to the table *Store*. The *Store* table must exist when the *Inventory* table is created. Thus, the order of the CREATE TABLE statements produced by the schema translator is important. However, it is not always possible to determine if a table must be created before other tables. This is the case when a table is involved in a referential cycle as shown in figure 3.12.

In this figure, *Table1* refers to *Table2*, which refers to *Table3*. *Table3* refers to *Table1*, and thus it cannot be determined which table must be created first. This problem can be solved by omitting the definition of the foreign keys in the produced CREATE TABLE statements. Hence, the various tables can be created in any order. When the tables have been created, the foreign key definitions, that have been left out of the CREATE TABLE statements,

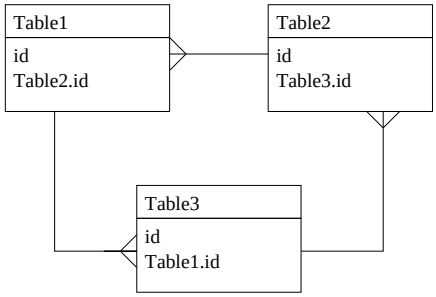


Figure 3.12: Tables involved in a referential cycle

must be inserted into their respective tables. This can be achieved by the ALTER TABLE statement [SKS97].

Translation of explicit constraints
Explicit constraints are expressed in the logical design phase using the *set constraint* statement from the SnowLanguage. The *set constraint* statement is used for expressing the constraints derived from the conceptual design phase. The *set constraint* statement is used for three different purposes in order to reflect the ACs, SACs, and ECs from the conceptual design phase:

- For constraining the domain of an attribute. These constraints are derived from ACs.
- For specifying how a summarizable attribute must be aggregated. Such constraints are derived from SACs.
- For constraining the row instances (tuples) of a table. These constraints are derived from ECs.

The *set constraint* statement has the following syntax:

set constraint *identifier* **for** (*table_id* | *attr_id*) ":"
(*aggr_expr* | *ranges*)

The keywords of the *set constraint* statement are written in boldface. The first *identifier* is the name of the constraint. The keyword **for** is followed by either a *table_id* or an *attr_id*. The *table_id* is used for specifying a table, and the *attr_id* is used for specifying an attribute on a table. The semicolon

is followed by either an *aggr_expr* (aggregation expression) or *ranges*. The *aggr_expr* is used when the constraint definition is used for specifying how an attribute must be aggregated. The *ranges* are used for two purposes. If the constraint definition is specified for an attribute on a table, the *ranges* is used for defining the domain of the specified attribute. If the constraint definition is specified for an table, the *ranges* are used for defining a condition for a table.

A *set constraint* statement that is used for constraining the domain of an attribute in a table, is translated into a CONSTRAINT clause [Ora99b]. The CONSTRAINT clause is useful for constraining the domain of an attribute, where the domain must depend on other attribute values. By specifying a CHECK clause together with the CONSTRAINT clause, the CONSTRAINT clause expresses a condition that must always be satisfied. That is, whenever new data is loaded into the data warehouse, the new data is accepted only if it does not violate the constraint defined by the CONSTRAINT-CHECK clause.

In the following we provide an example of how a constraint definition on an attribute domain is translated into SQL statements. The name of the constraint definition is *acLoanAmount*. The constraint definition specifies that the value of the *amount* attribute on the *Loan* table must be lesser than 10000, if the value of the *age* attribute on the *Person* table is lesser than 25.

```
set constraint acLoanAmount for Loan.amount:
    Loan.amount < 10000 if Person.age < 25;
```

Figure 3.13 shows the relation between the *Loan* and the *Person* tables, which are related through the *Repayment* table.

Figure 3.13: The relation between the Loan, Repayment and Person tables

In order to translate the *acLoanAmount* constraint definition into a CONSTRAINT-CHECK clause, the NOT-EXISTS clause [Ora99b] is used in the CHECK clause. The NOT-EXISTS clause is used for specifying that the result of a query must be empty i.e., no rows exist in the result returned by a query. This is useful, as this can be used for forming a query that extracts data on from the *Loan* and *Person* tables about the values of the *amount*

and *age* attributes. Thus, the *acLoanAmount* constraint definition can be translated into the following SQL constraint clause:

```
CONSTRAINT acLoanAmount CHECK
    (NOT EXISTS
        (SELECT * FROM Person WHERE age < 25 and ssn =
           (SELECT ssn FROM Repayment WHERE id =
               (SELECT id FROM Loan WHERE NOT amount < 10000))));
```

The innermost query in the constraint clause extracts all *id*s from the *Loan* table where the *amount* is not lesser than 10000. These *id*s are used for extracting all the *ssn* (Social Security Number) values from the *Repayment* table, where the *id* of the *Repayment* equals the extracted *id*s from the innermost query. The outermost query uses the extracted *ssn* values from the subquery in order to extract all tuples in the *Person* table, where the age is lesser than 25 and the *ssn* equals an *ssn* value extracted in the subquery. Thus, the NOT-EXISTS clause is used for checking if any tables have been extracted from the outermost query. If this is the case, the constraint definition is violated.

A *set constraint* definition that is used for specifying how an attribute must be aggregated is translated into the SQL statement: SELECT-FROM. This statement is used for forming one or more queries, where the extracted data is aggregated. The SQL aggregate functions AVG, COUNT, MAX, MIN, and SUM are used for aggregating data extracted from the queries.

In the following, we provide an example of how a *set constraint* definition, used for specifying how an attribute must be aggregated, is translated into SQL statements. The following constraint definition must be translated into SQL statements:

```
set constraint sacSumValue for Store.stock_value:
    sum(Inventory.value)
```

The name of the constraint definition is *sacSumValue*, and is used for identifying the constraint definition. The constraint definition is specified for the attribute *stock_value* on the *Store* table. The *stock_value* must be aggregated by summing all the values of the *value* attribute on the *Inventory* table. This constraint definition is translated into the following SQL statements:

```
UPDATE Store SET stock_value =
    (SELECT SUM(value) FROM Inventory);
```

The UPDATE-SET statement is used for updating the value of the attribute *stock_value* in the *Store* table. The SELECT-SUM-FROM query sums the values of the *value* attribute from the *Inventory* table.

A *set constraint* definition that is used for constraining row instances (tuples) is translated into the SQL statement: CREATE TRIGGER. The CREATE TRIGGER statement is an Oracle SQL specific statement [Ora99b], which is used for creating a *trigger*. A *trigger* is a statement that is executed automatically by the DBMS as a side effect of a modification to the database [SKS97]. A trigger specifies the conditions under which the trigger is to be executed, and the actions to be taken when the trigger executes [SKS97]. The CREATE TRIGGER statement is available in SQL *Plus, and can be used for constraining an entire row in a table.

In the following, we provide an example of how a *set constraint* definition, used for constraining the row instances of a table, is translated into SQL statements. The following constraint definition must be translated into SQL statements:

```
set constraint ecPersonAge for Person: Person.age > 18;
```

The name of the constraint definition is *ecPersonAge*. This constraint is defined on the *Person* table, and specifies that row instances (tuples) in this table can be inserted or updated, if the value of the *age* attribute is greater than 18. This *ecPersonAge* constraint definition is translated into the following SQL statement:

```
CREATE TRIGGER ecPersonAge BEFORE UPDATE OF age ON Person
    FOR EACH ROW ecPersonAgeProc;
```

This trigger statement specifies that the procedure *ecPersonAgeProc* will be called, before an update of the *age* attribute on the *Person* table occurs. The FOR EACH ROW clause [Ora99b] specifies that the *ecPersonAgeProc* procedure will be called for each row in the table that is affected by an update operation on the *age* attribute.

Note that the *ecPersonAgeProc* is a procedure that is not generated by the translator. It is the task of the designer to provide the procedure for the trigger.

3.6 The Repository

The *Repository* component of the design tool architecture (see figure 3.1) is the component that store schemas in the AWARE DESIGN TOOL.

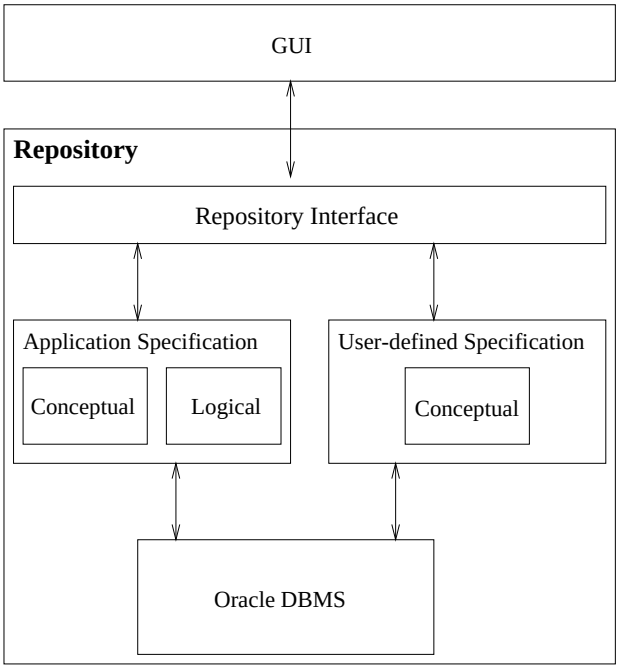


Figure 3.14: Architecture of the Repository.

The *Repository* component consists of the following sub-components (see figure 3.14):

Repository Interface: This part of the *Repository* provides access to the functionality of the *Application Specification* sub-component and *User-defined Specification* sub-component.

Application Specification: This component is split into two sub-components, as seen in figure 3.14. These sub-components perform the storing and loading of schemas. Each of the sub-components provide functionality for a specific design phase. Thus, there is a sub-component for both the conceptual and the logical design phase.

User-defined Specification: This component handles storage of user defined components. In the current implementation of the AWARE DE-

SIGN TOOL, the *User-defined specification* component has one sub-component, namely for the conceptual design phase (see figure 3.14). Thus, it is possible to store and load user-defined components from the conceptual design phase only.

Two storage schemas has been defined for the *Repository*: a *Conceptual storage schema* and a *Logical storage schema*. The *Conceptual storage schema* represents the starER model. The *Conceptual storage schema* can be seen in figure 3.15.

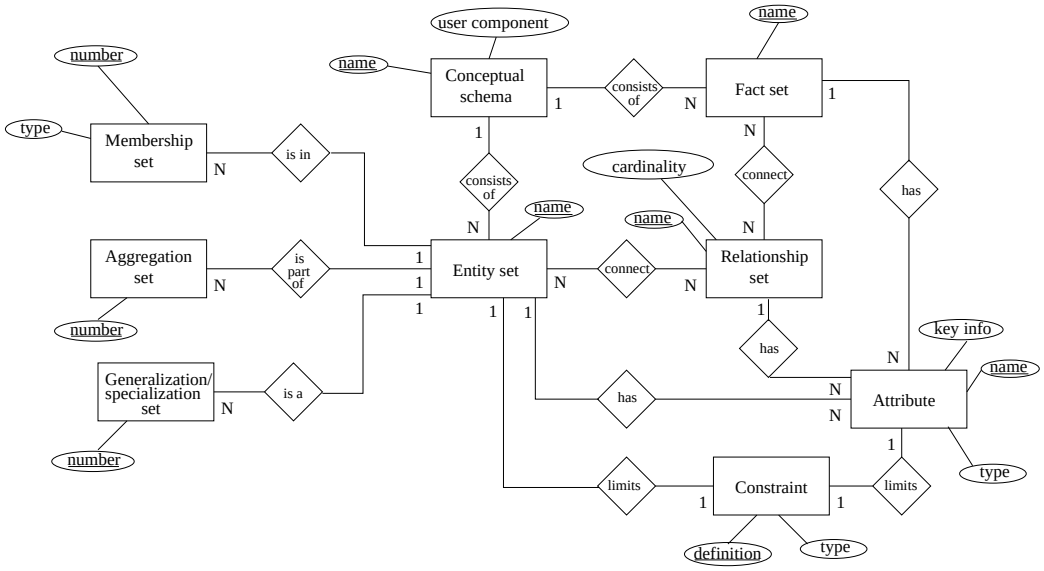


Figure 3.15: ER diagram of the Conceptual storage schema.

The elements in the *Conceptual storage schema* will be described in the following. Emphasized words refer to specific sub-components in the *Conceptual storage schema*.

- The entity set *Conceptual schema* represents definitions of conceptual schema (starER schemas). For a conceptual schema, the *name* of the schema is stored along with data about whether the schema is a user-defined component or not.
- The entity set named *Entity set* represents definitions of entity sets in a conceptual schema. For an entity set, the *name* of the entity set is stored.

- The entity set *Fact set* represents definitions of fact sets in a conceptual schema. For a fact set, the *name* of the fact set is stored.

- The entity set *Relationship set* represents definitions of relationship sets in a conceptual schema. For a relationship set, the *name* of the relationship set is stored along with data describing the *cardinality* of the relationship.

- The entity set *Membership set* represents definitions of membership sets in a conceptual schema. A membership set is defined on the entity set which is the member of another entity set. For a membership set, it is necessary to store the *type* (strict, complete or non-complete) and a *number* to identify the membership set.

- The entity set *Generalization/specialization set* represent definitions of generalizations/specialization sets in a conceptual schema. A generalization/specialization set is defined on the specialized entity set. For each generalization/specialization set, it is necessary to store a *number* that uniquely identifies the generalization/specialization set.

- The entity set *Aggregation set* represents definitions of aggregation sets in a conceptual schema. An aggregation is specified on the entity set which is a part of another entity set. For aggregation sets, a *number* is stored to uniquely identify the aggregation sets.

- The entity set *Attribute* represents definitions of attributes in a conceptual schema. For attributes, it is necessary to store the *name* of the attribute along with data about the *type* (Stock, Flow, Value-per-unit or regular attribute) and what *key type* (foreign, primary, or not a key) the attribute is.

- The entity set *Constraint* represents constraint definitions in a conceptual schema. For constraints, it is necessary to store the *definition* of the constraint. This defintion is unique because it contains the name of the entity set or attribute it is defined upon.

Figure 3.16 shows the ER diagram of the *Logical storage schema*. As can be seen in figure 3.16, the *Logical storage schema* is simpler than the *Conceptual storage schema*. The *Logical storage schema* allows the *Repository* to store data about logical schemas (Extended Snowflake schemas).

The elements in the *Logical storage schema* are described in the following.

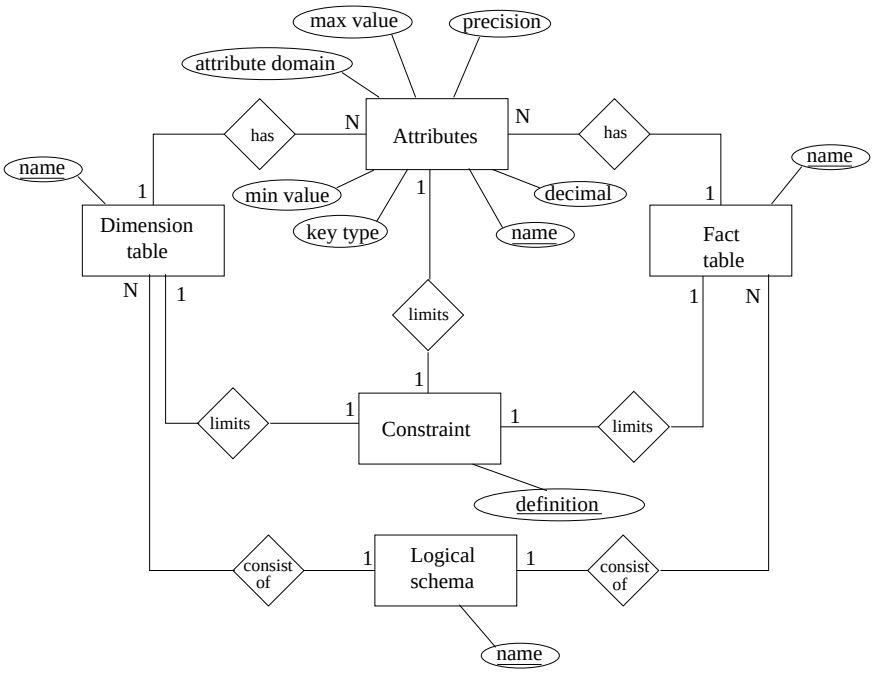


Figure 3.16: ER diagram of the Logical storage schema.

- The entity set *Logical schema* represents the definition of a logical schema (an Extended Snowflake schema). For logical schemas, it is necessary to store the *name* of the schema. A logical schema consists of *Fact table*s and *Dimension table*s.

- The entity set *Fact table* represents definitions of fact tables in a logical schema. For fact tables, it is necessary to store the *name* of the fact table.

- The entity set *Dimension table* represents dimension tables in a logical schema. As with fact tables, it is necessary to store the *name* of the dimension table. Both dimension and fact tables has *Attributes* defined upon them.

- The entity set *Attributes* represents definitions of attributes in a logical schema. For attributes, it is necessary to store their *name* and data about whether the attribute is a key (foreign or primary) or not a key. In addition, it is necessary to store data about the *domain* (varchar, integer etc.) of the attribute. Along with this data, it is necessary to store data about the *precision*, number of *decimals*, *min value* and *max*

value for the attribute.

- The entity set *Constraint* represents constraints defined on either *Dimension tables*, *Fact tables* or *Attributes*. For constraints, it is necessary to store the *definition*. This definition contains the *name* of the constraint, and is therefore sufficient as primary key for a constraint.

In this chapter, we have described the components in the AWARE DESIGN TOOL. It is our conclusion, that the architecture, as described in section 3.1, has proved to be an advantage when implementing the AWARE DESIGN TOOL, as it made the implementation of the sub-components of the AWARE DESIGN TOOL easier. Regarding the communication among the components in the architecture, we realize that the coupling between the GUI and other components in the AWARE DESIGN TOOL is too strong. This makes the GUI component inflexible regarding changes to the AWARE DESIGN TOOL. We conclude that the metadata containers are very useful regarding communication among components in the AWARE DESIGN TOOL. The parsers and generators, as described in section 3.4, are necessary in order to convert graphical schemas to source code and vica versa. The translators are necessary in order to support the methodology, as described in chapter 2, as they make it possible to translate a schema from one design phase to the next. We also conclude, that a database is suitable for storing data about schemas. In the next chapter, we propose different expansions to the AWARE DESIGN TOOL.

4 Expanding the Aware Design Tool

In this chapter, ideas for expanding the functionality of the Aware Design Tool are presented. First we shortly describe how schemas are currently maintained in the Aware Design Tool, and we describe how they should be maintained. Next we present our idea for a *Controller*. We describe how the *Controller* can be used for maintaining constraints, and we give an example of how the *Controller* should handle structural changes to a schema. Then we present different ideas of how the *Controller* could be used regarding the Aware Design Tool. These ideas include the description of a conceptual query language, and how the *Controller* could be used to implement a data security administration in the data warehouse design tool.

4.1 Maintaining Schemas

When explicit conceptual constraints (ACs, SACs and ECs) are defined in a conceptual schema, an important issue is how these constraints are maintained in the logical schema. This issue also applies when constraints are defined for a schema, either conceptual or logical, and the schema structure is changed subsequently.

In the Aware Design Tool, constraints are currently handled in a static manner. This means that explicit conceptual constraints are translated into constraint definitions in the logical schema (see appendix C for constraint

translations). Moreover, if the structure of a schema is changed or if an element in a schema is renamed, the changes are not reflected in the constraint definitions. This is a problem, because when renaming occur, the constraints must still be imposed on the elements upon which they were defined. In order to overcome this problem, it must be possible to maintain constraints in a dynamic manner. That is, when an element is renamed, any constraint that is imposed, on the element being renamed, must be updated to reflect the change.

Regarding constraint definitions, a problem occur if an attribute is referenced in a constraint definition, and this attribute is either moved or renamed. This is a problem because it would cause inconsistency within the constraint definitions. Such changes must be reflected in the constraint definitions which references the attribute. To overcome this problem, it should be possible to handle structural changes in a dynamic manner. For example, when an attribute is moved, all constraint references to this attribute must be changed.

Another problem is how to handle conceptual constraints when the conceptual schema has been translated into a logical schema. If a constraint is imposed on either an entity set or an attribute, this constraint must be imposed on the corresponding table or attribute in the logical schema. For an EC, this means that the table, corresponding to the entity set upon which the EC was defined, cannot be deleted. That is, the entity set cannot be deleted, as this would violate the EC definition from the conceptual design phase that must be maintained in the logical design phase. Moreover, it should not be possible to delete or move any attribute in this table. For ACs and SACs, this means that the attribute in the logical schema corresponding to the attribute upon which the AC or SAC was defined cannot be deleted. In fact, we suggest that in a logical schema, no attribute which originates from a conceptual schema can be deleted

In order to overcome the above described problems, we introduce the idea of a *Controller*. The *Controller* must ensure that a schema is always consistent regarding references. The *Controller* can achieve this by propagating changes throughout a schema whenever they occur. The functionality and the structure of the *Controller* is described in the next section.

4.2 The Controller

In order to overcome the problems listed in the previous section, we suggest a *Controller*. First we define the notion of a *Controller*.

Controller: *A component which enforces that constraints are maintained dynamically in a schema.*

The *Controller* should be integrated as the other components in the AWARE DESIGN TOOL architecture (see section 3.1). We suggest that the *Controller* consists of three distinct sub-components (see figure 4.1). These are described in the following.

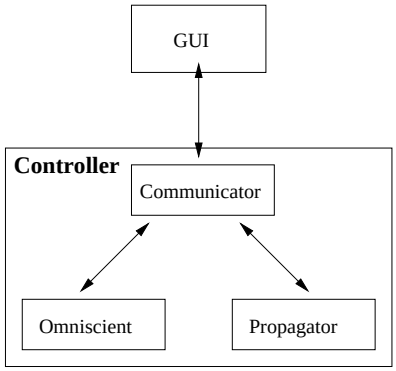


Figure 4.1: Structure of the Controller

Propagator: *A component that propagates changes throughout a schema.*

The functionality of the *Propagator* can be achieved by letting the *Propagator* use the metadata containers (see section 3.3) in order to maintain the schema structure. When changes occur to the schema, the *Propagator* changes the contents of the metadata containers where necessary, and thereby changing the schema. This means that the GUI component must use the contents of the metadata containers maintained by the *Propagator* to reflect the current schema state.

Omniscient: *A component that is used for determining whether a change to a schema is allowed or not.*

The *Omniscient* component holds information about which constraints are imposed on a schema, whether it being explicit, implicit or inherent constraints, these constraints are described in [NLK99]. Also, the *Omniscient* holds information about which implicit constraints can be deducted from the

definition of explicit constraints. This information is used to check whether a change to a schema can be allowed or not. Because the *Omniscient* holds information about inherent constraints, this implies that the *Omniscient*, and thereby the *Controller*, is dependent upon the used data model. Therefore, a *Controller* must be implemented for each design phase supported in the AWARE DESIGN TOOL.

Communicator: *A component that handles communication among the sub-components in the Controller and the GUI.*

The purpose of the *Communicator* component is to send requests to the *Omniscient* and, if the request is granted, to inform the *Propagator* about which changes must be made. If a request is not granted, then the *Omniscient* informs the *Communicator* about this decision, and the *Communicator* returns an error message to the GUI.

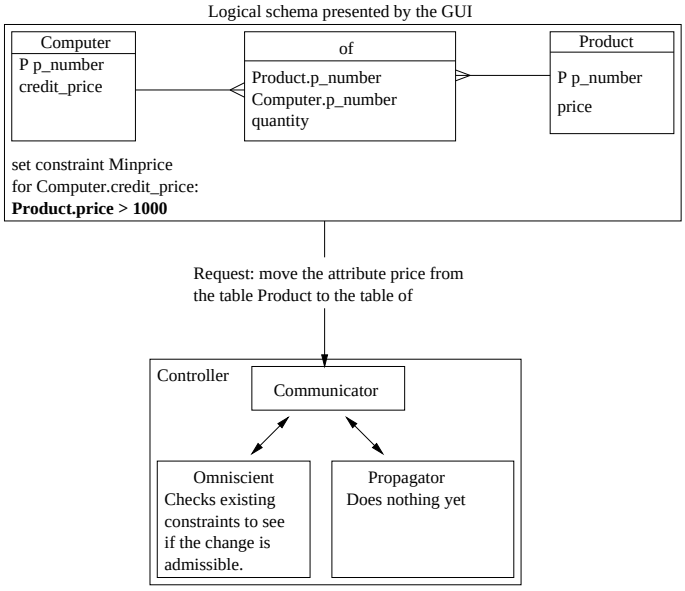


Figure 4.2: The GUI request permission to make a change.

To clarify how the *Controller* handles changes to a schema structure, an example from the *Sales data warehouse* logical schema is presented in figure 4.2 (see [KLN00] for more information about the Sales case study). This example describes how structural changes to the *Product*[1] dimension should

[1]Note that elements in the dimension has been left out since they are not of interest in this example.

be handled dynamically by the *Controller*.

The changes we want to impose on the schema structure is complete denormalization of all dimensions. This means that the tables *Product* and *Computer* should be collapsed into the *of* table. This change has effect on the constraint *Minprice*, which references the attributes *Computer.credit_price* and *Product.price* (see figure 4.2).

The most interesting step in this context is to move the attribute *price* from the *Product* table into the *of* table. This results in a request being send to the *Communicator* from the GUI (see figure 4.2). The *Communicator* pass the request on to the *Omniscient*. The *Omniscient* checks if any constraint definitions that would make the action impossible.

In this example, no such constraint is defined. Therefore, the *Omniscient* concludes that the change is admissible and inform the *Communicator* about this (see figure 4.3). The *Communicator* then prompts the *Propagator* to make the necessary changes to the schema. The *Propagator* then traverses the metadata containers and perform changes where ever they are necessary in the schema. Finally, the *Propagator* orders the GUI to rebuild the schema (see figure 4.3). Such action will be performed everytime the designer attemps to make a change to the schema.

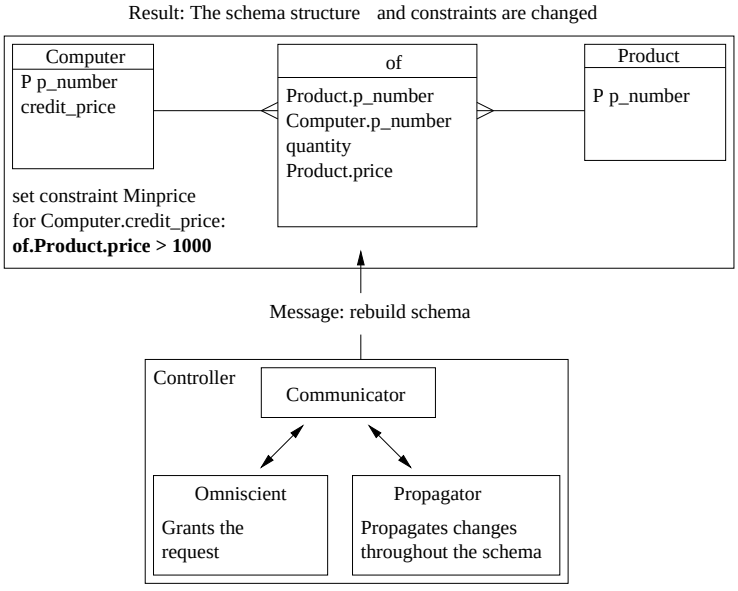


Figure 4.3: The request is granted.

In the next section, we suggest to expand the AWARE DESIGN TOOL, so it is possible to administrate security at the conceptual design phase.

4.3 Data Security

Due to legislation regarding security for data registers and because companies may not allow their data warehouse users to query upon all data in the data warehouse, data security is needed for data warehouses. By data security, we mean that it should be possible to hide data from certain user groups. For example, sales managers should not be able to query about customers' personal information (social security number, income etc.). We suggest that security should be administrated at the conceptual design phase. This makes it possible for a company manager to participate in the administration along with the data warehouse designer, using concepts familiar to the manager. In order to handle security administration in the AWARE DESIGN TOOL, we suggest a *Guardian*.

Guardian: *A component that ensures that certain data in a data warehouse is accessable to selected user groups only.*

The *Guardian* must be able to create the necessary protection mechanisms for the data in the data warehouse. In addition, the *Guardian* must be able to assign user groups to parts of the data warehouse schema and to create user groups.

Because the AWARE DESIGN TOOL permits the designer to change the structure of logical schemas (see chapter 2), the *Guardian* should be able to deal with such changes dynamically. That is, the restrictions on user access to the data in the data warehouse must be consistent regardless of the schema structure. The *Controller*, as proposed in section 4.2, handles schema changes dynamically. Thus the *Guardian* should be notified about changes in the schema by the *Controller* in order to provide the necessary functionality that ensures that the restrictions of the user access will be consistent, even when changes are made to the schema structure of the data warehouse.

Figure 4.4 shows how the GUI and the Controller should communicate with the Guardian.

Handling administration of user groups in the AWARE DESIGN TOOL can be achieved by using an administrative window in the GUI. In this window, it should be possible to administrate user groups. This should be done in

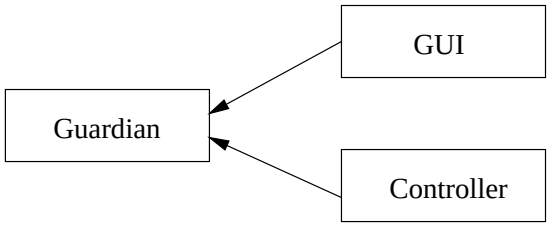


Figure 4.4: The GUI and the Controller communicating with the Guardian.

a similar manner as Oracle Security Manager, which permits the database administrator to maintain profiles (see [Ora99a]).

To clarify how the *Guardian* should handle security administration, an example from the Sales data warehouse is presented (see [KLN00] for elaboration on the Sales case study). In this example, store managers should only be able to query upon sales regarding the store they manage. The first step in this example is to create a *Store Manager* user group. The next step is to restrict the *Store Manager* user group by assigning elements in the schema relevant to this user group (see figure 4.5).

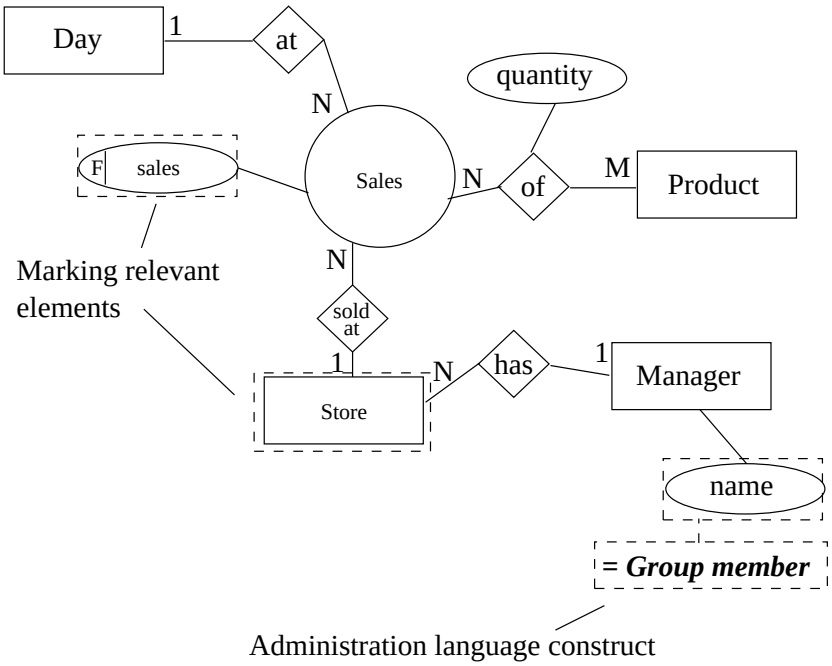


Figure 4.5: Restricting data access for Store Managers.

In order to express how user groups can be restricted to certain parts of data in the data warehouse, we suggest a conceptual administration language. The *Group member* could be a construct in such a conceptual administration language (see figure 4.5). The *Group member* construct is used to ensure that the group members (Store Managers) are restricted to queries that involves the store they manage. A conceptual administration language should also include constructs that makes it possible to express other restrictions. For example, it would be desirable to include a construction that can be used to restrict user groups to query upon parts of the data. For example, the Store Manager group should be restricted to only query upon products, where product number equals 12.000.

Recall that when the conceptual schema has been designed, the conceptual schema is translated into a logical schema which may be modified. When the logical schema is modified, inconsistency in the user group restrictions can occur. Therefore, when modifications are made to the logical schema, the *Controller* must notify the *Guardian* about the changes. The *Guardian* should then make the necessary changes in order to ensure that the user group access rights are consistent with the current schema structure.

Finally, when the logical schema is translated into SQL statements, the *Guardian* outputs an additional set of SQL statements. These SQL statements are used to enforce the security restrictions in the underlying DBMS.

4.4 Conceptual Query Language

In this section, we provide a description of ideas for developing a conceptual query language. This conceptual query language should make it possible to specify queries at a conceptual level, based on the starER model as presented by [TBC99]. We propose a high-level graphical query language that allows end-users to specify queries on a starER schema in an intuitive way. This query language should make it possible for the end-users of the data warehouse to extract and view data from the data warehouse graphically.

The AWARE DESIGN TOOL does not support a conceptual query language, as this is not the purpose of this tool i.e, the AWARE DESIGN TOOL is used only for designing data warehouses. Therefore, we propose that an additional tool is developed, which makes data mining possible i.e., a data mining tool. In the following, we refer to this tool as the *Aware Query Tool*. When a data warehouse has been designed using the AWARE DESIGN TOOL, and data has been loaded into the data warehouse, the *Aware Query Tool* should be used for querying upon the data in the data warehouse.

In order to specify queries using the *Aware Query Tool*, the starER schemas must be provided for the *Aware Query Tool*. These starER schemas are available in the repository of the AWARE DESIGN TOOL (see 3.6). Thus, the repository in the AWARE DESIGN TOOL should be accessible from the *Aware Query Tool*. Note that the *Aware Query Tool* should not be allowed to modify the schemas stored in the repository, as modifications to a data warehouse schema is a design issue.

A query can be specified on a starER schema by *selecting* elements in the schema. A *selected* element is an element that will participate in the query, and is marked by a thick border around the element on the schema. If the selected element is an attribute, this attribute will participate in the query. If the selected element is an entity set, relationship set, or fact set, *all* of the attributes defined on the selected element will participate in the query. Note that it should be possible to select several attributes defined on the same element individually.

It should be possible to constrain the values of a selected attribute that participates in a query. This is achieved by defining a *query constraint* on the selected attribute.

Query constraint: *A condition that limits the query result.*

A query constraint is specified on a schema by a dashed box that is connected

to the selected attribute.

In the following, we provide an example of how a graphical query can be specified on a conceptual schema. This example is based on the Sales data warehouse as described in [KLN00]. Note that in order to keep this example as simple as possible, only the *Time* and *Product* dimensions are considered. In the example, we want to extract all sales for a specific range of products in a specific year. More specific, we want to extract all sales in 1999 for products that have a product number between 10.000 and 13.000. Figure 4.6 shows how such a query could be specified on a starER schema. In this figure, the following elements are selected:

- The attribute *year* on the *Year* entity set in the *Time* dimension.
- The attribute *p_number* on the *Product* entity set in the *Product* dimension.

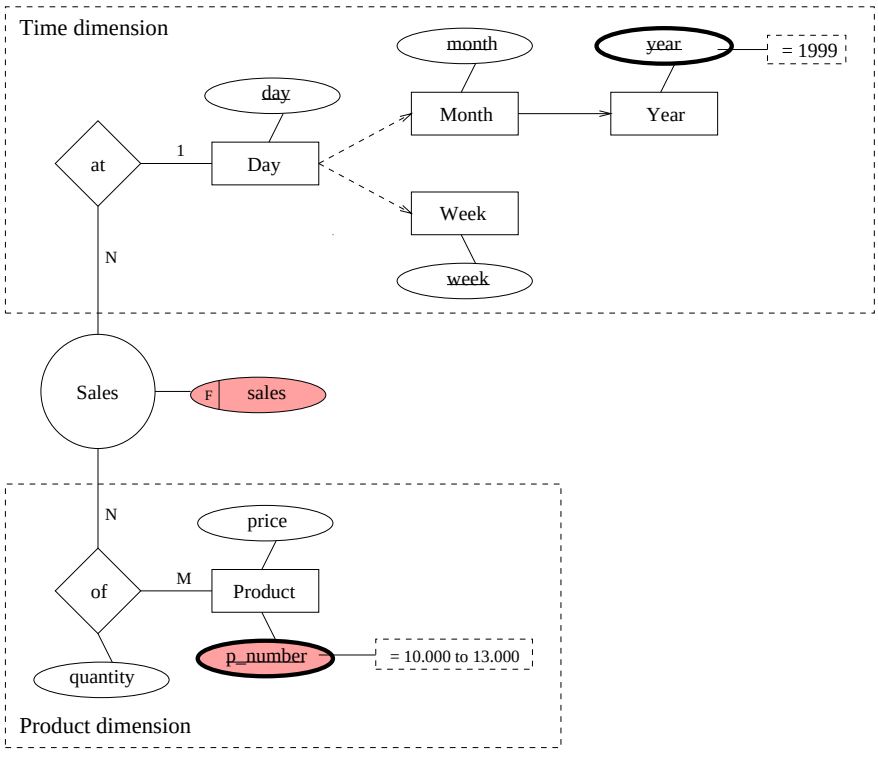


Figure 4.6: A conceptual query.

In order to specify that only sales from year 1999 must be considered in the query, a query constraint is defined for the *year* attribute on the *Year* entity

set (see figure 4.6). This query constraint specifies that the value of the *year* attribute must be equal to 1999. That is, an expression, "= 1999", is written within the query constraint. Thus, the query is constrained to only return values where the *year* attribute equals the value 1999.

Similarly, a query constraint is defined on the *p_number*, which is used for specifying that only products with a product number (*p_number*) in the range of 10.000 to 13.000 are considered in the query. In this case the expression, "= 10.000 to 13.000", is written within the query constraint (see figure 4.6).

By marking elements using a color, it is possible to specify what elements the query should return. In figure 4.6, the following elements are marked by a color, and thus defines the result of the query:

- The summarizable attribute *sales* on the *Sales* fact set.
- The regular attribute *p_number* on the *Product* entity set in the *Product* dimension.

The result of the query in the example is shown in table 4.1. This table consists of two columns containing the values of the marked attributes in tabel 4.1 i.e., the *p_number* and *sales* attributes.

Product.p_number	**Sales.sales**
10.000	2.710
10.001	5.117
10.002	7.589
...	...
12.998	4.165
12.999	3.584

Table 4.1: Sales *of* Product.p_number = 10.000 to 13.000 *at* Year.year = 1999.

Another aspect of a graphical query language is the ability to specify that extracted data in a query must be aggregated. For example, we might want to sum the sales of all the product numbers for 1999 returned by the query shown in figure 4.1. This can be achieved by specifying a *Sum* function[2] for the *sales* attribute on the *Sales* fact set using a dashed box as shown in figure 4.7. The result of using the Sum function in the query is shown in table 4.2.

[2]Note that the graphical query language should also support other aggregation functions.

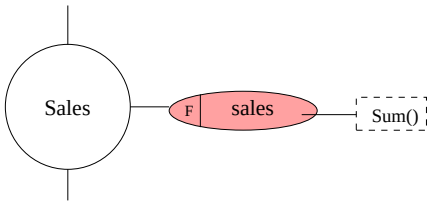


Figure 4.7: Specifying an aggregation query.

We suggest that a conceptual query specified using the graphical query language is translated into a SQL query at the logical level in the *Aware Query Tool*. This SQL query should then be used for extracting data in the data warehouse. In order to achieve the translation into a SQL query, the *Aware Query Tool* must contain a query translator. This query translator must be able to identify the query paths that exists among selected elements in a query. That is, the query translator must be able to join the elements that exists in a query path, in order to make the translation of the query into SQL possible.

Summed Sales.sales
14.253.789

Table 4.2: Summed sales *of* Product.p_number = 10.000 to 13.000 *at* Year.year = 1999.

The query translator must have access to data regarding the mapping between the conceptual schema to the logical schema. This is a requirement, as the elements and structure in the conceptual schema could be very different from the resulting logical schema. That is, first the conceptual schema has been translated into a logical schema by the AWARE DESIGN TOOL. Secondly, the structure of the logical schema, and the names of the elements in this schema may have been changed during the logical design phase. Thus, by accessing data regarding the mapping between the conceptual schema and the resulting logical schema, the translator should be able to map the conceptual query into a logical query, i.e. a SQL query.

In this chapter, we have described problems regarding schema changes. The AWARE DESIGN TOOL does not handle these problems in the current implementation. In order to overcome these problems, we have suggested a Controller. A Controller is useful if the AWARE DESIGN TOOL should be able to dynamically handle changes to a schema. Also, a Controller compo-

nent is necessary in order to implement the proposed Guardian that should be able to handle the administration of security at the conceptual design phase. Finally, we have suggested to implement a conceptual query language.

5 Conclusion & Future Work

In this chapter we conclude on the work described in this report, and we suggest future research issues.

5.1 Conclusion

In this report, we have described a data warehouse design methodology. It is our conclusion, that a design methodology provides the designer with several advantages when designing a data warehouse. In order to utilize these advantages, we have implemented the Aware Design Tool to support the methodology. We conclude that it is possible to implement a design tool that supports such a methodology, and that the advantages of the methodology are retained in the Aware Design Tool.

Regarding the architecture of the Aware Design Tool, we can conclude that defining this architecture has been an advantage. The architecture has made it easy to split the actual implementation of the Aware Design Tool into smaller parts. Moreover, the architecture has made it easy to test the implementation of components in the architecture individually. Also, the architecture provided makes it easy to replace any component when necessary, as well as adding new components.

We conclude that the metadata containers implemented in the Aware Design Tool has proven very useful to us. They are useful because they make

the communication between the other components in the AWARE DESIGN TOOL easy. Also, the metadata containers can be utilized when expanding the functionality of the AWARE DESIGN TOOL. This is an advantage because it is possible to let a component maintain the metadata containers, and make the GUI reflect the contents of the containers.

Regarding the schema translators, we conclude that these are necessary in order to translate a schema from one design phase to the next automatically. Furthermore, we conclude that it is necessary to perform the translation of the schema elements in a specific order to ensure that the translation is performed correctly.

An advantage of using a database for storing schemas in the repository is that it is possible for other tools to utilize the schemas, and the schemas are easily ported to other platforms.

5.2 Future Work

Regarding future work we suggest that the *Controller*, as described in section 4.2, is implemented in the AWARE DESIGN TOOL. The *Controller* should be an important part of the AWARE DESIGN TOOL, but because of the limited time at our disposal, there was not time enough to implement this component.

When the *Controller* is implemented, we suggest that the areas of a conceptual query language and conceptual security administration are explored. A conceptual query language should be implemented in a separate data mining tool, but utilize the functionality of the *Controller* and the *Repository* of the AWARE DESIGN TOOL.

Another suggestion for future work is the ability of specifying how data in the data warehouse should be pre-aggregated. Such functionality should be implemented at the logical design phase in the AWARE DESIGN TOOL.

Another expansion of the AWARE DESIGN TOOL functionality could be to allow associating data from external data sources with elements in the logical schema. This could be used to semi-automatically create a data fetching/cleansing component.

Finally, It would be desirable to extend the AWARE DESIGN TOOL into a CASE tool. This would require the functionality of e.g., project management and version control of schemas.

A The StarLanguage Syntax

This appendix provides the syntax of the StarLanguage using the Backus Naur Form (BNF) notation [Guy00]. Keywords are written in boldface, and terminals that consists of only one or two characters are surrounded by quotes (").

schema ::=
    schema identifier **is** definition { definition } | { definition }

definition ::=
    fact | entity | relationship | ec | ac | sac

fact ::=
    fact identifier properties "."

entity ::=
    entity identifier entity_spec ".";

entity_spec ::=
    isa **and** properties | properties

isa ::= **is** is_spec

is_spec ::=
    a identifier_list | **part of** identifier_list | membership_list

identifier_list ::=
    identifier { "," [**and**] identifier }

membership_list ::=
    membership { "," [**and**] membership }

membership ::=
    membership_type membership_spec

membership_type ::=
    complete | **non-complete** | **strict**

membership_spec ::=
    member of identifier

relationship ::=
    relationship identifier relationship_spec ".";

relationship_spec ::=
    connections [**and** properties]

connections ::=
    connects connection "," [**and**] connection_list

connection_list ::=
    connection { "," [**and**] connection }

connection ::=
    identifier [cardinality_spec]

cardinality_spec ::=
    with cardinality cardinality

cardinality ::=
    one | **many**

properties ::=
    has attributes [**and has** position] | **has** position [**and has** attributes]

attributes ::=
    attributes attribute { "," [**and**] attribute }

attribute ::=
    identifier attribute_spec [**at** position]

attribute_spec ::=
    as type attribute_type

attribute_type ::=
    regular | **key** | **stock** | **flow** | **value-per-unit**

position ::=
    position point

point ::=
    "(" value "," value ")"

ec ::= **ec** identifier ":" expr

ac ::= **ac** attr_id ":" domain_expr_list "."

sac ::= **sac** attr_id ":=" aggr_expr_dim_list

expr ::=
    log_expr | comp_expr | arit_expr | aggr_expr | value_expr | "(" expr ")"

log_expr ::=
    not expr | expr **and** expr | expr **or** expr

comp_expr ::=
    expr comp_op expr

arit_expr ::=
    expr arit_op expr

aggr_expr ::=
    aggr_op "(" expr ")"

value_expr ::=

value | identifier | attr_id

domain_expr_list ::=
value { ";" domain_expr }

domain_expr ::=
[expr ":"] domain

domain ::=
range { "," range }

range ::=
value [**to** value] | comp_expr

aggr_expr_dim_list ::=
aggr_expr [**per** identifier_list]

identifier ::=
single_word_id | quoted_id

attr_id ::=
identifier "." identifier

value ::=
digit { digit } ["." digit { digit }]

single_word_id ::=
letter { letter | digit }

quoted_id ::=
""" { any_character } """

comp_op ::=
"=" | "<>" | "<" | ">" | "<=" | "=>"

arit_op ::=
"+" | "−" | "*" | "/"

aggr_op ::=
sum | **avg** | **min** | **max** | **count**

B The SnowLanguage Syntax

This appendix provides the syntax of the SnowLanguage using the Backus Naur Form (BNF) notation [Guy00]. Keywords are written in boldface, and terminals that consists of only one or two characters are surrounded by quotes (").

stmt_list ::=
    [stmt { ";" stmt_list }]

stmt ::= **set** set_stmt

set_stmt ::=
    fact_table_stmt | dim_table_stmt | constr_stmt

fact_table_stmt ::=
    facttable table_spec

dim_table_stmt ::=
    dimtable table_spec

constr_stmt ::=
    constraint identifier **for** (identifier | attr_id) **":"** (aggr_expr | ranges)

table_spec ::=
identifier attr_list **and** key_list **at** "(" number "," number ")"

attr_list ::=
with attr { "," [**and**] attr }

key_list ::=
primary_key_list { "," [**and**] key_list } | foreign_key_list { "," [**and**] key_list }

primary_key_list ::=
primary key_identifier_list

foreign_key_list ::=
foreign key_reference_list **references** identifier

key_identifier_list ::=
key "{" identifier_list "}" | **key** identifier

identifier_list ::=
identifier { "," [**and**] identifier }

key_reference_list ::=
key "{" reference_list "}" | **key** reference

reference_list ::=
reference { "," [**and**] reference }

attr ::=
identifier **as** attr_type [**not null**]

attr_type ::=
integer | **real** | **numeric** "(" integer "," integer ")" | **float** "(" integer ")" |
varchar "(" integer ")" | **date** | **time**

expr ::=
log_expr | comp_expr | arit_expr | aggr_expr | value_expr | "(" expr ")"

ranges ::=
range_cond_opt { "," range_cond_opt }

log_expr ::=

not expr | expr **and** expr | expr **or** expr

comp_expr ::=
expr comp_op expr

arit_expr ::=
expr arit_op expr

aggr_expr ::=
aggr_op "(" expr ")" [dim_identifier_list]

value_expr ::=
number | identifier | attr_id

range_cond_opt ::=
range [**if** expr]

range ::=
number ["−" number] | comp_expr

dim_identifier_list ::=
per "{" identifier_list "}" | **per** identifier

identifier ::=
single_word_id | quoted_id

reference ::=
identifier "->" identifier | identifier

attr_id ::=
identifier "." identifier

integer ::=
digit { digit }

number ::=
integer ["." integer]

single_word_id ::=
letter { letter | digit }

quoted_id ::=
"""" { any_character } """"

comp_op ::=
"=" | "<>" | "<" | ">" | "<=" | ">="

arit_op ::=
"+" | "−" | "*" | "/"

aggr_op ::=
sum | **avg** | **min** | **max** | **count**

C Translation Rules

The translation rules are used for translating a starER schema into an Extended Snowflake schema. The translation rules ensure that all components in a starER schema are translated properly into components in an Extended Snowflake schema. Moreover, constraints that are defined in the starER schema are translated into constraints in the the Extended Snowflake schema.

The translation is performed in 11 steps. The first 8 steps are used for translating all components in a starER schema, i.e., fact sets, entity set, relationship sets etc. The last 3 steps are used for translating constraint definitions in the starER schema. The translation rules must be followed step by step in the order they are presented in order to obtain a valid Extended Snowflake schema.

A primary key must be present in all the dimension tables in an Extended Snowflake schema. This is necessary as all dimension tables in the Extended Snowflake Schema must be referenced to by another dimension or fact table. Thus, if no primary key is provided for a dimension table it is necessary for the translation rules to provide a primary key for the dimension table.

Relationship sets are handled in two separate steps. First one-to-many relationship sets are handled. Secondly, many-to-many and high-order relationship sets are handled. The two steps are required in order to translate fact sets properly. Fact sets in a starER schema are translated into fact tables for the Extended Snowflake schema. In order to translate a many-to-many or high-order relationship set between a fact set and an entity set, a primary key must exist on the fact table in the Extended Snowflake schema corresponding to the fact set in the starER schema. The primary key of a fact table is composed entirely of all its foreign keys. Thus, these foreign

keys must first be included into the fact table before the many-to-many and high-order relationship sets can be properly translated. This is achieved in the translation step for the one-to-many relationship set. This translation step includes foreign keys in fact tables.

Summarizable attributes are handled in a separate step, as all summarizable attributes on entity sets and relationship sets are translated into fact tables [NLK99]. However, this is not the case for summarizable attributes defined on fact sets. Summarizable attributes defined on fact sets in the starER schema are included in the corresponding fact table in the Extended Snowflake schema.

During the translation of a starER schema into an Extended Snowflake schema, it can be necessary to rename an attribute that is about to be included in a table in the Extended Snowflake schema. This is necessary only if an attribute already exists in the table, which has the same name as the attribute that is about to be included in the table.

In the following subsections, the translation steps are described.

Fact Sets

In this initial step a new fact table is created in the Extended Snowflake schema for each fact set that exists in the starER schema. All regular and summarizable attributes on the fact set in the starER schema are included in the corresponding fact table in the Extended Snowflake schema.

Translation Rule:

For each fact set F in the starER schema, create a new fact table T_F in the Extended Snowflake schema.

- Include regular and summarizable attributes of F as attributes of T_F.

Entity Sets

In this step, entity sets from the starER schema are translated into dimension tables for the Extended Snowflake schema, except for entity sets that are a subpart in an aggregation set. Entity sets that are a subpart in an aggregation set are handled separately in the step used for translating aggregation sets.

All regular attributes on an entity set in the starER schema must be included in the corresponding dimension table in the Extended Snowflake schema. Note that summarizable attributes are not included in the dimension tables, as these are handled separately in the step used for translating summarizable attributes.

If a primary key is defined on an entity set in the starER schema, this key is used as the primary key in the corresponding dimension table in the Extended Snowflake schema. If no primary key is defined on the entity set, then a primary key must be provided for the corresponding dimension table in order to ensure that the dimension table has a primary key.

Translation Rule:

For each entity set E in the starER schema that is not a subpart of an aggregation set, create a new dimension table T_E in the Extended Snowflake schema.

- Include all regular attributes of E as attributes of T_E.
- If a primary key K is specified on E, then K becomes the primary key of T_E; else create a new primary key K and include this key in table T_E.

Aggregation Sets

No dimension table should be created in the Extended Snowflake schema for entity sets in the starER schema that are subparts of an aggregation. Instead, all regular attributes from these entity sets (subparts) are included in the dimension table in the Extended Snowflake schema that corresponds to the aggregated entity set in the starER schema. At this point of the translation, this dimension table has already been created in the step used for translating entity sets.

Translation Rule:

For each aggregated entity set A in the starER schema, identify the corresponding dimension table T_A in the Extended Snowflake schema, and identify all entity sets $E_1, E_2, \ldots, E_n$ that are subparts of A in the starER schema.

- Include all regular attributes from each entity set E_i into T_A.

One-to-many Relationship Sets

No dimension table is created in the Extended Snowflake schema in order to represent a one-to-many relationship set from the starER schema. Instead the table in the Extended Snowflake schema corresponding to an entity set or fact set at the *many*-side of a relationship set in the starER schema is used for representing the one-to-many relationship set. In this table the primary key of the table at the *one*-side of the relationship set is included as a foreign key. If the relationship set has regular attributes, these are included in the table that is used for representing the relationship set.

Translation Rule:

For each one-to-many relationship set R in the schema, identify the table T_M in the Extended Snowflake schema that corresponds to the entity set or fact set at the *many*-side of the relationship set. Let the table in the Extended Snowflake schema that corresponds to the entity set on the *one*-side in the starER schema be T_1. If the one-to-many relationship set has a generalized entity set at the *many*-side in the starER schema, then identify the dimension tables $t_{S_1}, t_{S_2}, \ldots, t_{S_n}$ that corresponds to the specialized entity sets of this generalized entity set.

- Include the primary keys of T_1, $t_{S_1}, t_{S_2}, \ldots, t_{S_n}$ as foreign keys in T_M.
- If the relationship set R has regular attributes then include these in T_M.

Many-to-many and High-order Relationship Sets

In order to represent a many-to-many or high-order relationship set from the starER schema, a new dimension table must be created in the Extended Snowflake schema. The primary keys of all the related tables in the Snowflake schema corresponding to the related entity sets and/or fact sets in the starER schema are included in this dimension table as foreign keys. If the relationship set has regular attributes, these are included in the dimension table that represents the relationship set.

Translation Rule:

For each many-to-many or high-order relationship set R in the starER schema that relates the entity sets or fact sets $X_1, X_2, \ldots, X_n$ in the starER schema, create a dimension table T_R in the Extended Snowflake schema. If the relationship set is related to one or more generalized entity sets in the starER schema, then identify the dimension tables $t_{S_1}, t_{S_2}, \ldots, tS_n$ that corresponds to the specialized entity sets of the generalized entity sets.

- Include the primary keys $K_1, K_2, \ldots, K_n$ of the entity sets or fact sets $X_1, X_2, \ldots, X_n$ as foreign keys in the dimension table T_R.
- Include the primary keys of $t_{S_1}, t_{S_2}, \ldots, tS_n$ as foreign keys in T_R.
- The primary key of T_R is the combination of all the included foreign keys.
- If the relationship set R has regular attributes then include these in T_R.

Specializations

When translating entity sets that are specializations of a generalized entity set in the starER schema, the dimension table corresponding to the generalized entity set and each dimension table corresponding to the specialized entity sets in the Extended Snowflake schema must be identified. All the attributes and foreign keys that are defined on the dimension table corresponding to the generalized entity set, must be included in each of the dimension tables that corresponds to the specialized entity sets.

In the step used for translating entity sets, a primary key has already been provided for all dimension tables that have been created in the Extended Snowflake schema. A specialized entity set in the starER schema inherits the primary key from its super classes (generalized entity sets). Thus, the primary key of the dimension tables in the Extended Snowflake schema corresponding to specialized entity sets in the starER schema must be replaced with a new primary key. This new primary key is composed of all the primary keys from the dimension tables in the Extended Snowflake schema corresponding the specialized entity set's super classes in the starER schema.

Translation Rule:

For each specialized entity set S in the starER schema, identify the dimension tables $T_{G_1}, T_{G_2}, \ldots, T_{G_n}$ in the Extended Snowflake schema that corresponds to the super classes of entity set S in the starER schema.

- Include all the regular attributes from $T_{G_1}, T_{G_2}, \ldots, T_{G_n}$ as new attributes of S.
- Include all the foreign keys from $T_{G_1}, T_{G_2}, \ldots, T_{G_n}$ as new foreign keys of S.
- Replace the primary key of S with a new primary key that is composed of all the primary keys from $T_{G_1}, T_{G_2}, \ldots, T_{G_n}$.

Membership Sets

In this step, dimension tables in the Extended Snowflake schema that corresponds to entity sets that take part in membership sets in the starER schema are handled. This is achieved by identifying the dimension table corresponding to the entity set of finer granularity, and then including the primary key of this table into the dimension table corresponding to the entity set of coarser granularity as a foreign key.

Translation Rule:

For each membership set M in the starER schema, identify the dimension tables T_{E_i} and T_{E_j} that takes part in M, which corresponds to the member entity set E_i of coarser granularity and the member entity set E_j of finer granularity from the starER schema.

- Include the primary key K_{E_i} of T_{E_i} as a new foreign key in the dimension table T_{E_j}.

Summarizable Attributes

A summarizable attribute on an entity set or relationship set in the starER schema is translated into a fact table in the Extended Snowflake schema. The primary key of the dimension table corresponding to the entity set or

relationship set on which the summarizable attribute is defined, must be included as a foreign key in this fact table.

A summarizable attributes is aggregated over one or more hierarchies. This is specified on the summarizable attribute by an aggregation expression. The aggregation expression specifies an entity set from each of the hierarchies, that the summarizable attribute is aggregated over. Each of the specified entity sets from the hierarchies are used for defining the granularity of the aggregation over a specific hierarchy. Thus, the primary key of each of the dimension tables corresponding to an entity set from the specified hierarchies must be included as foreign keys in the fact table, which have been created for the summarizable attribute.

By default a summarizable attribute will use the granularities of the entity set or relationship set in the starER schema, which has the summarizable attribute. However, if a Summarizable Attribute Constraint (SAC) is specified for the summarizable attribute, which specifies other granularities, then the summarizable attribute must use these granularities instead.

Translation Rule:

For each entity set or relationship set X in the starER schema that has a summarizable attribute A, identify the corresponding dimension table T_X in the Extended Snowflake schema.

- Create a new aggregate fact table T_A for the summarizable attribute A.
- The primary key K_X of T_X is included in T_A as a foreign key.
- Include the primary keys $K_1, K_2, \ldots, K_n$ as foreign keys in T_A from the dimension tables $T_1, T_2, \ldots, T_n$ corresponding to the member entity sets in the starER schema, which specifies the granularities of A.
- The primary key of T_A is composed of all foreign keys of T_A.

Entity Constraints (ECs)

An Entity Constraint (EC) definition in the StarLanguage is translated into a *set constraint* statement for the Extended Snowflake schema. An EC definition is specified on an entity set, and its condition is expressed using regular

and/or summarizable attributes. The entity set and the attributes, which are specified in the EC definition have been translated into tables and attributes for the Extended Snowflake schema by the previous translation rules. Thus, when an EC definition is translated into a *set constraint* statement, this statement must specify the translated tables and attributes in the Extended Snowflake schema, which corresponds to the entity set and attributes from the starER schema that is specified in the EC definition.

Translation Rule:

An EC definition for a starER schema is defined using the following StarLanguage syntax:

> **ec** *identifier* ":" *expr*

The *identifier* is used for identifying the entity set in the starER schema which the constraint definition is specified on. The *expr* is the expression used for specifying the condition of the constraint definition.

An EC definition is translated into a *set constraint* statement using the following SnowLanguage syntax:

> **set constraint** *constraint_id* **for** *table_id* ":" *expr* ";"

The *constraint_id* is used to identify the constraint definition in the Extended Snowflake schema. The *table_id* is used to indentify the table in the Extended Snowflake schema which this constraint definition is specified on. The *expr* is the expression used for specifying the condition of the constraint definition.

For each EC definition that is defined in the starER schema, create a *set constraint* statement for the Extended Snowflake schema and translate according to the following:

- Create an unique *constraint_id* for identifying the constraint definition in the Extended Snowflake schema.
- Identify the table in the Extended Snowflake schema that corresponds to the translated entity set from the starER schema that is specified by the *identifier* in the EC definition. Set the *table_id* of the constraint definition to identify this table.

- Translate the expression of the EC definition from the starER schema into a corresponding expression for the Extended Snowflake schema.

Attribute Constraints (ACs)

An Attribute Constraint (AC) definition for an attribute is translated into a *set constraint* statement in the Extended Snowflake schema. An AC definition is specified for an attribute, and its condition is expressed using regular and/or summarizable attributes. The attributes which are specified in the AC definition have been translated into attributes on tables for the Extended Snowflake schema by the previous translation rules. Thus, when an AC definition is translated into a *set constraint* statement, this statement must specify the translated tables and attributes in the Extended Snowflake schema that corresponds to the entity sets, fact sets and atributes from the starER schema, which are specified in the AC definition.

Translation Rule:

An AC definition for a starER schema is defined using the following StarLanguage syntax:

ac *attr_id* ":" *expr*

The *attr_id* is used for identifying the attribute in the starER schema, which the constraint definition is specified on. The *expr* is the expression used for specifying the condition of the constraint definition.

An AC definition from a starER schema must be translated into a *set constraint* statement for the Extended Snowflake schema using the following SnowLanguage syntax:

set constraint *constraint_id* **for** *attr_id* ":" *expr* ";"

The *constraint_id* is for identifying the constraint definition in the Extended Snowflake schema. The *attr_id* is used for indentifying the attribute for which this constraint definition is specified on. The *expr* is the expression used for specifying the condition of the constraint definition.

For each AC definition that is defined in the starER schema, create a *set constraint* statement for the Extended Snowflake schema and translate according to the following:

- Create a unique *constraint_id* for identifying the constraint definition in the Extended Snowflake schema.
- Identify the attribute in the Extended Snowflake schema, which corresponds to the translated attribute from the starER schema that is specified by the *attr_id* in the AC definition. Set the *attr_id* of the constraint definition for the Extended Snowflake schema to identify this attribute and the table it is defined on.
- Translate the expression of the AC definition from the starER schema into a corresponding expression for the Extended Snowflake schema.

Summariable Attribute Constraints (SACs)

A Summarizable Attribute Constraint (SAC) definition expressed in the StarLanguage for a summarizable attribute is translated into a *set constraint* statement from the SnowLanguage for the Extended Snowflake schema. A SAC definition is specified for a summarizable attribute, and its condition is expressed using regular attributes and/or other summarizable attributes. The attributes which are specified in the SAC definition have been translated into attributes in the Extended Snowflake schema by the previous translation rules. Thus, when an SAC definition is translated into a *set constraint* statement, this statement must specify the translated tables and attributes in the Extended Snowflake schema, which corresponds to the entity sets, fact sets and atributes from the starER schema that is specified in the SAC definition.

Translation Rule:

An SAC definition for a starER schema is defined using the following StarLanguage syntax:

sac *sum_attr_id* ":=" *aggr_expr*

The *sum_attr_id* is used for identifying the summarizable attribute in the starER schema, which the constraint definition is specified on. The *aggr_expr*

is the aggregation expression used for specifying how the summarizable attribute must be aggregated.

An SAC definition from a starER schema must be translated into a *set constraint* statement for the Extended Snowflake schema using the following SnowLanguage syntax:

set constraint *constraint_id* **for** *attr_id* ":" *aggr_expr* ";"

The *constraint_id* is for identifying the constraint definition in the Extended Snowflake schema. The *attr_id* is used for indentifying the attribute in the Extended Snowflake schema which this constraint definition is specified on. The *expr* is the expression used for specifying the condition of the constraint definition.

For each SAC definition that is defined in the starER schema, create a *set constraint* statement for the Extended Snowflake schema and translate according to the following:

- Create a unique *constraint_id* for identifying the constraint definition in the Extended Snowflake schema.
- Identify the attribute in the Extended Snowflake schema, which corresponds to the summarizable attribute from the starER schema that is specified by the *sum_attr_id* in the SAC definition. Set the *attr_id* of the constraint definition for the Extended Snowflake schema to identify this attribute and the table it is defined on.
- Translate the expression of the SAC definition from the starER schema into a corresponding expression for the Extended Snowflake schema.

Bibliography

[BCN92] Carlo Batini, Stefano Ceri, and Shamkant B. Navathe. *Conceptual Database Design.* The Benjamin/Cummings Publishing Company, inc., 1992.

[EN94] Ramez Elmasri and Shamnkant B. Navathe. *Fundamentals of Database Systems.* The Benjamin/Cummings Publishing Company, inc., second edition, 1994.

[Guy00] J. Guyot. What is bnf notation? http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html, May 21st 2000.

[Kel99] Thomas J. Kelly. Dimensional data modeling. Available on-line at:http://www.gate1.com/solutions/whitepapers/sybase/syb_dim_data_mod.html, 27th october 1999.

[KLN00] Peter S. Kristiansen, Flemming N. Larsen, and Carsten Nielsen. The aware design tool, a user guide. Master's thesis, Aalborg University, 2000.

[Mat96] Rob Mattison. *Data Warehousing.* McGraw-Hill, 1996.

[MMMNS97] Lars Mathiassen, Andreas Munk-Madsen, Peter Axel Nielsen, and Jan Stage. *Object Orienteret Analyse og Design.* Forlaget Marko Aps, 1997.

[NLK99] Carsten Nielsen, Flemming N. Larsen, and Peter S. Kristiansen. Aware design tool, a data warehouse design tool. Dat5 report, CS department Aalborg University, 1999.

[Ora99a] Oracle. Oracle warehouse builder. Available on-line at http://www.oracle.com/datawarehouse/products/builder/index.html, 12th November 1999.

[Ora99b] Oracle Corporation. *Oracle Lite SQL Reference*, release 4.0 edition, 1999. Available on-line at http://technet.oracle.com/docs/products/8i_lite/doc_index.htm.

[SKS97] Abraham Silberschatz, Henry F. Korth, and S. Sudarshan. *Database System Concepts*. McGraw-Hill, third edition, 1997.

[TBC99] Nectaria Tryfona, Frank Busborg, and Jens G. Borch Christiansen. starer: A conceptual model for data warehouse design. *Proceedings of DOLAP'99*, 1999.