Formal Design of Data Warehouse and OLAP Systems

A dissertation presented in partial fulfilment
of the requirements for the degree of

Doctor of Philosophy
in
Information Systems

at Massey University, Palmerston North, New Zealand

Jane Qiong Zhao

2007

Abstract

A data warehouse is a single data store, where data from multiple data sources is integrated for online business analytical processing (OLAP) of an entire organisation. The rationale being single and integrated is to ensure a consistent view of the organisational business performance independent from different angels of business perspectives. Due to its wide coverage of subjects, data warehouse design is a highly complex, lengthy and error-prone process. Furthermore, the business analytical tasks change over time, which results in changes in the requirements for the OLAP systems. Thus, data warehouse and OLAP systems are rather dynamic and the design process is continuous. In this thesis, we propose a method that is integrated, formal and application-tailored to overcome the complexity problem, deal with the system dynamics, improve the quality of the system and the chance of success.

Our method comprises three important parts: the general ASMs method with types, the application tailored design framework for data warehouse and OLAP, and the schema integration method with a set of provably correct refinement rules.

By using the ASM method, we are able to model both data and operations in a uniform conceptual framework, which enables us to design an integrated approach for data warehouse and OLAP design. The freedom given by the ASM method allows us to model the system at an abstract level that is easy to understand for both users and designers. More specifically, the language allows us to use the terms from the user domain not biased by the terms used in computer systems. The pseudo-code like transition rules, which gives the simplest form of operational semantics in ASMs, give the closeness to programming languages for designers to understand. Furthermore, these rules are rooted in mathematics to assist in improving the quality of the system design.

By extending the ASMs with types, the modelling language is tailored for data warehouse with the terms that are well developed for data-intensive applications, which makes it easy to model the schema evolution as refinements in the dynamic data warehouse design.

By providing the application-tailored design framework, we break down the design complexity by business processes (also called subjects in data warehousing) and design concerns. By designing the data warehouse by subjects, our method resembles Kimball's "bottom-up"approach. However, with the schema integration method, our method resolves the stovepipe issue of the approach. By building up a data warehouse iteratively in an integrated framework, our method not only results in an integrated data warehouse, but also resolves the issues of complexity and delayed ROI (Return On Investment) in Inmon's "top-down" approach. By dealing with the user change requests in the same way as new subjects, and modelling data and operations explicitly in a three-tier architecture, namely the data sources, the data warehouse and the OLAP (online Analytical Processing), our method facilitates dynamic design with system integrity.

By introducing a notion of refinement specific to schema evolution, namely schema refinement, for capturing the notion of schema dominance in schema integration, we are able to build a set of correctness-proven refinement rules. By providing the set of refinement rules, we simplify the designers's work in correctness design verification. Nevertheless, we do not aim for a complete set due to the fact that there are many different ways for schema integration, and neither a prescribed way of integration to allow designer favored design.

Furthermore, given its flexibility in the process, our method can be extended for new emerging design issues easily.

Acknowledgement

My sincere and huge thanks go to my supervisor Prof. Klaus-Dieter Schewe for all the guidance and support that he has given me generously during my study.

My thanks also goes to my co-supervisor Associate Prof. Roland Kaschek for his critical discussions on this work.

I am grateful towards my parents for their constant love and care, and the great help in looking after my children.

Finally my thanks to my partner, Henning Köhler, and my two beautiful and lovely daughters, Angela and Nicole, for their love and support which made my life meaningful and enjoyable.

It would be impossible for the completion of this work without any of the help from the people mentioned and the people around me.

Contents

List of Figures

Chapter 1

Introduction

A data warehouse is a single data store where data from multiple data sources is integrated for the purpose of online analytical processing (OLAP) in management decision support for an entire organisation. The rationale for being single and integrated is to ensure that the analytical statistics on the organisational performance are consistent and independent from different angles of business processes. This implies that a data warehouse needs to store the data for all the different subjects, such as finance, marketing, inventory, and so on, within the entire organisation. Furthermore, business requirements for OLAP systems change over time. Data warehouse and OLAP systems thus are rather dynamic and the design process is continuous.

This view of data warehousing was shared by most of data warehouse designers, specifically, the Data Warehousing Basics channel brought by DM Review and Kalido - the provider of adaptive enterprise data warehousing and master data management software [31], which says "As strategic as they are, enterprise data warehousing projects are highly complex and can be risky. Projects fail almost as much as they succeed, often because of long development cycles, poor information quality and an inability to adapt quickly to changing business conditions or requirements."

In summary, data warehouse and OLAP system design faces the following challenges:

- Overwhelming complexity due to the wide coverage of subjects;
- Error-prone due to the complexity of the process;
- Integrity issues due to the ever-changing user requirements.

Therefore, we need a design method that simplifies the design work, facilitates quality design, tackles the dynamics and at the same time ensures the integrity of the system.

A review of the literature has shown that due to the complexity of the design process, work reported so far in data warehouse and OLAP systems design has mainly focused on solutions which deal with isolated issues, such as:

- view selection, which considers a data warehouse as a set of computed database queries, i.e. the materialised views, based on the user queries (e.g. [38, 39, 41, 48]);
- the issue of selecting a set of views to materialise against a set of users queries considering query performance and view maintenance (e.g.[65, 4, 62, 80, 125]);

- the issue of how to maintain the materialised views when the databases, i.e. the data sources, change;
- data modelling, which considers data warehouse as a special database system, and concerns with what data model suits the data warehouse and OLAP (e.g. [5, 116]);
- schema design (e.g. [35, 83]);
- index design (e.g. [40]);
- conceptual design, which concerns with one of the design stages, (e.g. [111]);
- ETL (Extract-Transform-Load), which concerns on getting the the data from the sources, (e.g. [103]);
- dynamic design, which concerns adapting the design to the changes of user requirements (e.g. [109, 37, 61, 64, 9, 79, 19, 58]) and so on.

Development methods in the data warehouse industry such as the "top-down" approach by Inmon [53], and the "bottom-up" approach by Kimball [58] provide methods for the whole project life cycle. But both of the approaches have their focus on database schema design. The issues with the two approaches are well known [102]. The "top-down" approach is ideal but not practical due to the upfront cost and delayed ROI (return of investment). Whereas, the "bottom-up" approach has fast turn round but it may have the problem of system integrity[52]. Kimball uses the idea of conformed dimension to resolve the issue, but appropriate method and techniques are needed to support or enforce it.

Attempts have been made for constructing integrated and standard methods for the whole design process, such as in [76, 72]. Although the work in [76], which is close to our method, provides a MDA (Model Driven Architecture, an Object Management Group standard) oriented framework for the complete development process, it mainly focuses on the transformation of the models between the three tiers of the MDA and presents how multidimensional modelling is adopted using the Multidimensional Model Driven Architecture approach to produce multidimensional models, i.e. data warehouse schema design. It is not obvious how models at different layers of the data warehouse architecture are related, neither is how the system evolution is handled. The work in [72] presents a UML-based method for deriving a data warehouse from the underlying operational database following the "top-down" approach. The issues such as design complexity and delayed "ROI" are not addressed. The issues of UML as a modelling language are well discussed in [98, 17], among which we consider unclear semantics and unclarity with respect to abstraction level should particularly be addressed in the UML-based methods. These methods using UML packages may benefit from its support for an integrated framework, but without proper technique to attack the aforementioned problems, it may compromises the quality of the design.

Works dealing with dynamic issues of data warehouse so far are isolated from an integrated framework. These include reselection of views when the set of user queries changes, [109, 37, 61, 64], or adapting the data schema to the change of requirements, using schema evolution [9] or schema versioning [79].

In summary, as stressed in [89], very few comprehensive design methods have been devised so far. Issues in requirement analysis, schema evolution, distributed data warehousing, etc. need proper investigation.

My research aims at constructing an integrated design framework with a method for distributed, dynamic data warehouse and OLAP systems design which overcomes the issue of complexity and delayed ROI with the “top-down” approach and the issue of integrity with the “bottom-up” approach, and simplifies the design work, ensures the integrity of the system, improves the quality of the design and the chance of success.

First of all, we ground our approach in the method of Abstract State Machines (ASMs, [17]) for its ground model and stepwise refinement approach, which explicitly supports our view of systems development to start with an initial specification, i.e. the *ground model* [14] that is then subject to *refinements* [15]. So quality criteria such as the satisfaction of constraints can first be verified for the ground model, while the refinements are defined in a way that preserves already proven quality statements. In particular, this is assisted by a set of provably correct refinement rules. This approach is similar to the one taken in [97], which contains a refinement-based approach to the development of data-intensive systems using a variant of the B method [1].

Furthermore, the ground model method helps to capture the users’ requirements at such an abstract level that both domain experts and designers can understand, which implies that the domain experts can check by inspection if the ground model faithfully represents the requirements semantically, and the designers will be able to transform it correctly to an implementation without too much difficulty. The stepwise refinement method provides a general notion of refinement, which can be instantiated for specific purposes, e.g. schema refinement and strong schema refinement in our case, and gives freedom in selecting any proof methods, e.g. methods from other formal approaches, that work in a specific context, e.g. forward simulation from data refinement [27].

The choice of the ASMs method is made based on the success stories of the ASMs in many application areas, such as hardware and software architecture, databases, software engineering, etc. [16, 7, 84, 45, 49]. In particular, the mathematically sound ASM notion and easy-to-understand pseudo-code like transition rules [12], are reasons for the choice too. Of course, this does not conclude that other existing formal methods, e.g. Z, B, VDM, etc. are not capable of fulfilling the same goals. Specifically, many of them have been successfully used in real industry applications [21, 121], with many books published (e.g. [29, 33]) and intensive research carried out [20, 21, 32]. In fact, the ASMs method provides an open framework which allows the techniques from other formal methods to be borrowed whenever appropriate, e.g. the simulation approach in correctness of refinement proofs from the data refinement [27] is used in ASM refinement proofs [93].

Secondly, we extend the ASMs with types as we believe it is a good idea to incorporate types to make it closer to the well-established terminology in data-intensive systems, hence to make it easier to model the schema evolution by the notion of “schema refinement”, a notion of refinement to capture “schema dominance”. While there are several approaches to introducing typed versions of ASMs [28, 124], our rationale for introducing typed ASMs is far more modest, as our primary goal is to obtain an application-specific version of ASMs. Although the quality-assurance aspect is an important argument for a formal development method, there is still an aversion against using a rigid formal approach, in particular in areas, in which informal development is still quite dominant. So providing such a version of ASMs, i.e. the typed ASMs, that is easy to use due to the adoption of more familiar terminology will be a necessary step towards the simplification of systems development and the desired quality improvement.

Thirdly, we provide an integrated design framework which allows designers to break down the design process by subjects and design concerns such as new requirements, optimisation, and implementation. Our method allows a data warehouse to be built from a single subject iteratively. More subjects can be included whenever needed through the refinement under new requirements. The schema integration step in the method will integrate the schemata resulting from a new subject with the existing data warehouse schemata under the notion of schema dominance to preserve the capacity of the information and at the same time the integrity of the system. In such way we solve the integrity issue in the "bottom-up" approach.

By modelling the three tiers architecture, i.e. the data source, the data warehouses, and the OLAP (Online Analytical Processing), explicitly on both data (data schema) and operations, our method provides an integrated framework to ensure changes are propagated through the entire system from the OLAP, through the data warehouse, to the underlying operational database accordingly. In particular, we consider that modelling the OLAP, the focus of the decision support system, as part of the data warehouse system is necessary, as this is where the user requirements come from. This is how we can propagate the changes downward to the data warehouse, and then the data sources. Moreover, modelling the three-tier architecture facilitates the adaptation of changes from the data sources to the data warehouse as well.

Furthermore, we tackle the dynamics more completely, by schema evolution as in [9], and by view integration as in [19]. The former is adapted to the design of multidimensional databases, the later provides a method for materialised view data warehouse, i.e. a set of pre-computed and stored user queries or a set of commonly shared intermediate results of the user queries to be used as the data warehouse, whereas our approach is constructed for the widely used relational databases. In addition, we devise an algorithm for performance concern too. In that, our approach differs from the work in [109] by considering the case of adding new materialised views, when it is more beneficial to do so even if the new queries can be rewritten by the existing views.

Our approach facilitates the design by subject and design concerns which resolves the issues of complexity and delayed ROI (Return of Investment).

1.1 Contributions

The main contributions of this work are the following:

- We developed a comprehensive method for dynamic data warehouse and OLAP systems design which covers all the design concerns from the whole design process specific to data warehouse and OLAP systems design, and models a three-tier architecture including the data source, the data warehouse, and the OLAP system, and both data and operations explicitly.

- We simplified the design work by providing a dialect of ASM, the Typed ASM (TASM), so the well developed terminologies for data-intensive systems can be adopted easily, which is particularly helpful in modelling schema refinement in the schema integration.

- We introduced the notion of schema refinement to capture "schema dominance" in schema integration process, which provides the foundation for constructing the correct refinement rules for the schema integration rules.

- We constructed an integrated design framework with schema integration method which resolves the issues of complexity, delayed “ROI” and system integrity in data warehouse design.

- We developed a set of correctness proven refinement rules based on the notion of schema refinement and strong schema refinement for the schema integration which makes it easy for the designers to adopt the formal approach.

- Furthermore we took the performance concern into consideration in dynamic data warehouse design, by devising a view selection algorithm that overcomes some of the drawbacks in the existing work.

1.2 Outline

The remainder of the thesis is organised as follows:

Chapter two gives the review of the related work in the areas of data warehouse, schema integration and formal methods, with respective to system design;

In Chapter three, an detailed description of the Abstract State Machines (ASMs) is given based on the book [17] including the notions, semantics and syntactic constructs, and the ASM method;

In Chapter four, the ASMs are extended to incorporate a type system which is particularly designed for using the terms that are developed for data-intensive applications. The corresponding rules, terms, operations, and notion of refinements with types are introduced. Furthermore, the notion of schema refinement and the notion of strong schema refinement for schema integration are defined. A constructive proof on equivalence between a TASM and an ASM is presented;

In Chapter five, an ASM method-based and integrated framework for the design of data warehouse and OLAP systems is introduced. We show how the method facilitates the breaking down the design process by concerns such as new requirements, optimisation and implementation, and by subjects. The generalised ground model of data warehouse and OLAP systems is presented in the ASMs. In addition, ground models for a grocery store, as an example, are presented in both the ASMs and the TASMs. Furthermore, examples on how the ground model can be used to verify a set properties of the system are demonstrated, and some refinements specific to data warehouse and OLAP systems are presented with the refinement correctness proofs;

In Chapter six, we discuss view integration and cooperation as the method to integrate data warehouse schemata during data warehouse evolution. This includes an introduction on HERM (Higher-order Entity-Relationship Model) [106], and the notion of schema dominance and equivalence, and the presentation of a set of schema transformation rules with some guidelines on the integration process and the formulation of the corresponding refinement rules;

In Chapter seven, we present three case studies to show the application of the method and the refinement rules. These include dynamic system design, distribution design, and business statistics. In the case of dynamic design, we present an algorithm for materialised view selection which improves the existing method in dealing with dynamic design. In the case of distribution design, we introduce a query cost model and some heuristics on fragmentation;

Chapter eight contains the conclusion of the work and some indications for future developments.

Chapter 2

Literature Review

This chapter gives a review of the related work in the areas of data warehouse design, schema integration and formal system development methods.

2.1 Data Warehouse and OLAP Systems Design

A data warehouse is a single data repository where data from multiple data sources is integrated for online business analytical processing (OLAP) spanning the entire organisation. This implies a data warehouse needs to meet the requirements from all the business processes within the entire organisation. Thus, data warehouse design is a highly complex, lengthy and thus error-prone process. Furthermore, business analytical tasks change over time, which results in changes in the requirements for the systems. Therefore, data warehouse and OLAP systems are rather dynamic and the design process is continuous.

2.1.1 Design Methods

Since the emergence of data warehousing in the late 1980s, extensive research has been carried out. However, the majority of these efforts aim for solutions which deal with isolated issues, such as view selection, view maintenance, data modelling, conceptual design, schema design, index design, ETL (Extract-Transform-Load) modelling, dynamic data warehouse design, and so on.

Materialised Views as Data Warehouse

In the view selection approach, the data warehouse is seen as a set of materialised views defined over multiple data sources for answering a set of user queries. The simplest form of materialised view approach is to pre-compute and store all the user queries. A more effective way is to find a set of shared intermediate results of the queries, such that for a given storage space, it achieves an optimal solution to total query response time and view maintenance. Many works provide algorithms for view selection problems as shown in [38, 39, 41, 48].

The subsequent problem after the view selection is view maintenance. As the data is usually extracted from operational systems, sometimes called transaction systems, the data in these systems changes constantly. This implies that the selected views become out of date very quickly. To be useful, the materialised views need to be refreshed regularly. In particular, the views for OLAP analysis often involve historical data and so they are usually huge and the refreshing process takes time. Providing an optimal method for maintaining the materialised views has received intensive attention, such as [65, 4, 62, 80].

In addition to the changes in the data sources, the user requirements, e.g. the set of queries, change too. In the case that new queries are added, it is common that the selected materialised views cannot answer the new queries, so new views need to be materialised. In the case that existing queries are removed, some selected views are no longer useful, so they should be removed. [109] presents a method for finding a set of new views to materialise which with a given extra space, minimises the combined new query evaluation and new view maintenance cost. [107] presents a method for detecting the redundant views which can be removed without negative impacts on the existing query evaluation and view maintenance cost.

As the methods for view selection and maintenance focus on the methods of optimising query response and maintenance cost, they are never combined as an integrated design method for data warehouse and OLAP systems. Furthermore, the data warehouse designed as a set of materialised views cannot be used for OLAP analysis such as roll-up, drill-down, and drill-across. However, view selection and maintenance methods are useful as techniques for improving query performance in data warehouse systems design as shown in [6, 48, 61].

Design Methods in the Industry

Data warehouse design in the industry takes approaches different from view materialisation. It sees data warehouses as database systems with special needs, such as answering management related queries. The focus of the design becomes how the data from multiple data sources should be extracted, transformed, loaded (ETL in short) to and be organised in a database as the data warehouse. There are two dominant approaches, the “top-down” approach by Inmon [53] and the “bottom-up” approach by Kimball [58].

In the “top-down” approach, a data warehouse is defined as a subject-oriented,time-variant, non-volatile, integrated data repository for the entire enterprise [53]. Data from multiple sources are validated, reformatted, and stored in a normalized (up to 3NF) database as the data warehouse. The data warehouse will store “atomic” data, the data at the lowest level of granularity, from where dimensional data marts can be built by selecting the data needed for specific business subjects or specific departments. The “top-down” approach is a data driven approach as the data is gathered and integrated first and then business requirements by subjects for building data marts are formulated. The advantage of this approach is that it provides a single integrated data source, thus data marts built from it will have consistency when they overlap. The disadvantage to this method is that the initial effort, cost and time for implementing a data warehouse is significant. The development time from building the data warehouse to having the first data mart available to the users is substantial, leading to a late ROI (Return On Investment).

In the “bottom-up” approach, a data warehouse is defined as “a copy of transaction data specifically structured for query and analysis.” [58], namely the star schema. In this approach data marts are created first to provide reporting and analytical capabilities for specific business processes (or subjects). Thus it is considered to be a business driven approach in contrast to Inmon’s data driven approach. Data marts contain the lowest grain data and, if needed, aggregated data too. Instead of a normalised database for the data warehouse, a denormalised dimensional database is adopted to meet the information delivery requirements of data warehouses. Using this approach, in order to use the set of data marts as the enterprise data warehouse, data marts should be built with conformed dimensions in mind, meaning that common objects are represented the same in different data marts. The conformed dimensions link the data marts to form a data warehouse, which is usually called a virtual data warehouse. The advantage of the "bottom-up" approach is that it has quick ROI, as creating a data mart, a data warehouse for a single subject, takes far less time and effort than creating an enterprise-wide data warehouse. However, the independent development by subjects can result in inconsistencies, if common objects are not integrated and are updated at different times in the individual data marts. Although guidelines are provided in the method, such as a planning stage for data warehouse bus, it does not provide effective techniques, and thus it is not guaranteed that the method resolves the integrity problem in practice.

The work by Adamson et al [2] presents a collection of case studies, each of which shows how to build a dimensional model for a specific business process following the "bottom-up" approach. Their focus is to show, by presenting the case studies for simulating a common business company processes, such as sales, marketing, inventory, etc., how to build dimensional data models based on the business needs, and how to answer business queries using dimensional data models.

The work in [102] provides a comparison of data warehousing methodologies available in the industry, such as Oracle methodology, SAS methodology, etc., using a set of attributes they defined. Among these attributes, requirements modelling, data modelling and change management are particularly interesting. The requirements modelling attribute is used for measuring the techniques of capturing and modelling the business requirements, data modelling for data modelling techniques in developing logical and physical models, and change management for techniques in managing the changes in the data warehouse. According to their review, most data warehouse development methodologies put sufficient emphasis on modelling user requirements accurately. On the other hand, change management is an important issue of data warehouse design which is overlooked by most vendors in their methodologies. The finding in [102] indicate that none of the methodologies they reviewed has been recognized as a standard as yet.

Data Warehouse Modelling

Most data warehouse design processes consist of a set of common tasks, among which the question how to accurately and effectively model a data warehouse is considered to be important. As we take HERM (Higher-order Entity-Relationship Model) [106], an extended ER model with the convenience of defining relationship type over relationship types, as our visual conceptual model, and the star schema [58] is in fact of HERM type, methods for conceptual design is not our focus in this research work. Therefore, our discussion in this area does not aim to be exhaustive.

A common argument among those researchers who aim at creating new conceptual models for data warehouse design, is that conceptual design is critical in data warehouse design, but most design methods do not incorporate a conceptual design phase [111, 91]. Furthermore, they believe that the current conceptual model for database development, the ER model, does not capture the dimensions and facts explicitly, and thus is not suitable for data warehouse modelling. Considering a data warehouse as a special database, they suggest that normal forms for measuring good schema design should be defined and used.

The work in [111] claims that most data warehouse design methods, particulary those star schema-based, focus more on the logical and physical design of schema which lack the consideration of users' needs. Furthermore, they argue that the ER model, though widely used for conceptual modelling, is not suited for data warehouse design. Therefore, a model of higher abstraction, and more understandable to users than star schema, as the conceptual model, is needed. By identifying a set of concepts needed for modelling user requirements in data warehouse using an example of mortgage business, the work presents a model which combines ER model and data warehouse star schema into one single model, called starER, for incorporating the concepts.

The work in [91] suggests that conceptual modelling helps in achieving the flexibility and reusability of data warehouse schemata in the data warehouse evolution due to user requirements changing and getting extended over time. The work introduces a Multidimensional E/R Model (M E/R model) as a conceptual model in data warehouse design. Based on the fact that the ER modelling technique was created without multidimensional modelling in mind, and their belief that explicit dimension modelling is necessary, the authors present an extended ER model with constructs which are used to explicitly indicate if an entity or relationship is a dimension or a fact and the connections between the fact and dimensions. This work adds a dimensional view over the ER model.

The work in [51] proposes a systematic method for deriving a conceptual schema from operational databases, relational in particular. The authors consider a data warehouse as a special database so the design follows the same process for databases, such as requirement analysis, conceptual design, logical design and physical design. In particular, the conceptual design phase in data warehouse design has to identify dimensions, corresponding dimension hierarchies, and measures, and has to determine the allocation of the attributes from the underlying database(s) in the data warehouse schema. The authors assume the existence of a global ER schema for the data sources and rely on the business users to identify the measures and dimensions that are required in the data warehouse from the operational ER schema. They define a 3-phase process model for the conceptual design, through which the measures and the dimensions identified are gradually transformed into the graphical multidimensional schema, which consists of facts and a set of dimensions with hierarchies, in a normal form *MNF* designed for multidimensional databases in [68].

The work in [34] presents a graphical Dimensional Fact model as the conceptual model and a method to derive it from a global ER model of the operational database. This approach is similar to, but earlier than the work in [51]. The later uses normal forms in the identification of facts for improvements.

The work by Moody in [77] presents a method for developing dimensional models from ER models of the data sources. The three-steps process of the method involves data classification and dimension hierarchy identification using the operational ER model. The method follows the "top-down" approach, with the hope that the derived model is more resilient to future changes in user requirements.

The work in [112] present a model for OLAP application design which follows the "top-down" approach by assuming a data warehouse is built from the operational database first, before building a model for the OLAP analysis process. Therefore, the concepts for OLAP modelling captured in the MAC (Multidimensional Aggregation Cube), namely dimensions, dimension hierarchies, analysis paths, and measures, are closely related to OLAP operations, such as roll-up, drill-down, etc.

The work in [5, 116] defines data models for multidimensional databases using the concept of cubes, i.e. a multidimensional array. Agrawal et al [5] argue that the relational database model is for OLTP (On-line Transaction Process) not for OLAP. They define the cube-based model with a set of basic operations aiming for constructing OLAP functionalities. One nice thing about this model is that it treats dimensions and measures symmetrically. Instead of constructing OLAP functions from a set of basic cube operations, Vassiliadis [116] defines a set of operations which directly map the OLAP functionalities, such as roll-up, drill-down, etc. Both [5] and [116] present mappings of their models to relational databases, while [116] also describes mappings to multidimensional arrays.

The work in [82] introduce nine requirements the authors believe important in measuring a good multidimensional model. The authors further present a model which they claim addresses the nine requirements, among which there are the desired property symmetric treatment of dimension and measure, and other features such as recording uncertainty and different level of granularity in the model.

The work in [68] proposes two normal forms for defining "good schema" in the context of summarisability, a property describing the validity of aggregation on an attribute along a dimension or multiple dimensions with hierarchies. These normal forms are defined by restricting the existence of (weak) functional dependencies. The work also provides a more general normal form by introducing the concept of "context of validity" for attributes.

To summarise, we find most of the work so far merely provides models for combining ER model and dimensional model into one single model, for they believe both ER concepts and dimensional concepts are necessary in conceptual design for data warehouses. However, we believe that the purpose of a conceptual design phase is to map the user requirements to a first model of system design. Seeing how most of the methods try to derive the conceptual model from the underlying operational database, a data driven approach, it appears unlikely that these methods will build conceptual models that meet the user needs. This may be the reason why other data driven methods start from logical design onwards. Furthermore, a data warehouse derived from the operational database may not be able to meet the user requirements completely, since some requirements do have impacts on the underlying databases, e.g. to capture more data needed for user queries. They also may contain unwanted data, which makes them unnecessarily bulky.

Although the method in [51] does involve the users in the conceptual design, their approach of depending on the users in identifying the facts (i.e. measures) and dimensions appears impractical. My recent working experience at Massey University, New Zealand, in data warehousing has shown that an ER model of a data mart HEMI (Head counts and EFTS Management Information) attracts more business users attention than the star schema with dimensions and measures, which typically are not concepts within the grasp of business users.

Our understanding of the purpose for dimensional models are that they are good for effective information delivery, which is the centre of Kimball's argument [59] on why ER model, to be more specific, the 3NF of ER model, does not suit to data warehouse logical schema design. It is well known that the normalisation in relational database is meant for effectively storing information and maintaining its integrity in updates. However, it does not mean that the ER model in the conceptual level does not serve the purpose of conceptual modelling for data warehouse, a view also taken by Moody in [77].

Although we consider the ER conceptual model useful in data warehouse design, it does not suggest that the ER model is the only good model, or data models defined specifically for data warehouses have no merits. The models introduced in [5, 116] had little impact on design practice, since current database technology is dominated by the relational database model, but the symmetrical treatment of dimensions and measures is a desired property in dimensional modelling which provides the flexibility in OLAP query formulation. With asymmetrical treatment, a typical situation in relational star schema design is that new OLAP queries can result in redesign of the schema, e.g. when the new query groups attributes which were previously considered facts. The authors of [82] also consider the property of symmetric treatment to be important, and claim that their model has the property, but it is not clear in their report how it is achieved. Their discussion of how the model addresses the nine requirements they defined for good multidimensional modelling is far too brief.

Finally, the normal forms given in [68] for data warehouse schema design, which are based on the property of summerisability discussed in [69] provides techniques for solving the issue of validity of aggregation in dimensional database. They are particularly useful for complex multidimensional schema design, where normalisation algorithms are beneficial.

Work towards a Comprehensive and Integrated Design Method

After decades of development and application, data warehousing has become an important component in the management decision making process. However, as pointed out in [89], very few works provide designers with a design method for the whole design process within an integrated framework.

In [76], Mazon et al. present a method using Model Driven Architecture (MDA), an Object Management Group (OMG) standard framework that addresses the complete life cycle of software development for the whole data warehouse development process. For multidimensional modelling of data warehouses, they define a MD^{2}A (*MultiDimensional Model Driven Architecture*) approach to incorporate the multidimensional modelling concepts through the multi-tiers in the MDA. The method uses an extended UML as the formal way for building the models in different tiers, and the Query/View/Transformation (QVT) approach for the transformation of the models from the top tier, the computation independent model (CIM), downwards to platform independent model (PID), and to platform specific model (PSM). They adapt the data warehouse framework to the MDA which gives five layer (originated from the 5-tier architecture of data warehouse) and three view points (from the MDA). The MDA provides the support for system evolution through facilitating the changes in both the CIM and PIM, and then automatically realising them in the PSM.

In [72], Lujan-Mora et al. present a UML-based method to derives the data warehouse design from the underlying operational data sources. The method follows Inmon's "top-down" approach and covers the whole design of a data warehouse, which consists of four different schemas, the operational data schema, the data warehouse conceptual schema, the data warehouse storage schema, and the business model. The business model represents the data marts. To be an integrated approach, the method also includes the mappings from the operational database to the data warehouse conceptual schema, and from the data warehouse conceptual schema to the data warehouse physical schema.

An earlier work in [35] aims to provide a general methodological framework for data warehouse design, more specifically the schema design. The method defines a 6-phased design process, starting from the analysis of the source data with the output of a partial or whole schema as the global schema of the source data. It then moves to the requirement specification phase where the users will decide what will be the facts based on the global schema produced in the first phase. Then the conceptual design is to produce a dimensional model with facts and dimensions based on the global schema and the list of facts identified. The conceptual model is then refined with more details on workload and data volume in phase 4. In the logical design phase, the logical schema is produced to minimise the query response time, which is a combination of materialised views and a star schema. The final phase concerns the physical design of the schema using database techniques.

The above-mentioned UML-based methods benefit from the UML visual presentation in modelling and the support to being an integrated method for the whole system development, but if the issues of UML (discussed in [98, 17]), such as unclear semantics and inappropriate abstraction layer of user requirements, are not properly addressed, they can lead to ambiguities which compromise the correctness of the design. In neither of the works, these issues are addressed.

The method in [76] is similar to our method in that it models each layer of the five-layer data warehouse architecture with data and operations separated in different layers, whereas we model the three-tiers architecture with data and the related operations combined. The method also models three grand views representing a specific stage in the development life cycle, whereas in our method we have two grand views, the ground model and the refined models, the latter being a set of models each of which reflects a design decision and works as a formal documentation. As the paper focuses only on the multidimensional model design and the corresponding transformations of the models between the different views, it is unclear how ETL (Extract-Transform-Load) or system evolution are actually modelled. We consider it important for an integrated design method to demonstrate how the different parts of the design work as a whole.

We find the four-schema structure defined in [72] is unusual, as it is a mix of functionalities and design levels. The middle two schemas are for the data warehouse, one at the conceptual level, one at the physical level, while the other two schemas represent the underlaying operational data sources and the business model without saying which design level they are at. The authors leave out some important details which we consider are helpful for readers to understand their method, for example, the modelling of the mappings between schemas, which the authors also consider important for the method to be an integrated one.

The work in [35] focuses only on the schema design and it depends on the user in facts selection, which we find may not be practical.

2.1.2 Data Warehouse Evolution

As business conditions change constantly, the user requirements for a data warehouse never stay the same. Hence, it is natural that a data warehouse design method should deal with this dynamic issue. As agreed in [89], incorporating solutions for data warehouse evolution in the design method becomes an important issue in data warehousing. Unfortunately, few design methods reported so far include this important area. In the following, we will give a review on the works specific to data warehouse evolution.

Techniques and methods for data warehouse evolution have been reported in many works, such as [37, 109, 19, 61, 64], which are useful when data warehouse is implemented as a set of materialised views, or a set of materialised views is selected for improving the query performance in a data warehouse.

The work in [37] present a technique for effectively adapting the views when changes are made in the set of user queries. Instead of recomputing the changed query from the base data, they make use of the view definition of the current queries and compute the answer completely if possible, or partially from the result of the current queries locally. They call this problem of maintaining views by view redefinition the "view adaptation problem". This approach is particularly useful when query performance is critical, e.g. an interactive user query system. The technique is implemented by a set of algorithms to cater for the types of changes that can be made to a view in the form of an SQL query, with a cost-based query optimiser for optimal solutions in query performance.

In [19], a logical model for representing a data warehouse as a hierarchy of views of relational queries is proposed, namely the base views, the intermediate views, and the user views. The basic idea is to synthesise the data warehouse schema from a set of relational queries by first representing the queries using multiple query (evaluation) plans, and then integrating the set of multiple query plans into a single graph. Finally the single graph is pruned by removing redundancy. This logical model is used in dealing with the data warehouse evolution, classified by changes from the users, the data sources, and the materialised views. The last one is only relevant when views for query performance are included in the set of queries. This idea is somewhat similar to ours, in that integration is applied in their second view generation step. However, the paper does not provide technical details on the complex integration process.

In [109], the data warehouse is implemented as a set of materialised views over the source databases. The problem of incorporating new requirements is reduced to finding additional views to materialise for a set of new queries. The authors provide the technical details on the modelling of the problem and suggest algorithms and heuristics for solving it. Although the work provides a simpler method for coping with the changes than an approach which reselects views from scratch, with techniques to select such a set of new views that it is optimal w.r.t the set of new queries and the extra space given, this approach will not maintain an optimal design of the data warehouse w.r.t the complete set of the users queries in a long run, especially when considering this type of changes happens often.

In [64, 61], dynamic view selection is used in improving query performance. A system for dynamic view management is presented in [61]. By providing a dedicated space, the basic idea is to cache the aggregated query results during the user query time for answering subsequent user queries in the future. Discussions and techniques for how to effectively maintain the set of views are provided. The work in [64] suggests a two-phase operation for OLAP system, with startup phase for static view selection, and online phase for continues view selection. Their concern about the dynamics of view selection focuses on the change in query distribution, similar to the concern of a DBA (Database Administrator) in database tuning. In their approach, the statically selected materialised views will evolve from M_1 to M_2, ... during the online view selection phase. However, they provide few details on how the set of new views are actually derived during the online phase.

Evolution in materialised views as data warehouse has been well studied, and a few works dealing with schema evolution in dynamic data warehouse design are found in [58, 9, 79], which are not in the form of an integrated design method.

In Kimball's method [58], the issue of data warehouse change is focused on one aspect only, that is, for keeping the historical data in data warehouse when data from the source changes. Hence, the issue has only been addressed on dimensions, which he calls slowly changing dimensions, e.g. to capture a modified product in the product dimension. Sometimes the changes are so minor that the product will use the same product number. Kimball proposes three main techniques for handling this slow changing dimensions: overwriting, creating another dimension record, and creating a current value field. It is clear the first approach will not keep the historical data. The second approach is to modify the key with a version number attached to it. The last approach is to create a new field to capture the changes, but as Kimball suggested this approach should be used with care. The risk of data explosion may become an issue when such changes are frequent.

In [9], an algebra is defined for describing schema evolution and the effects on the instances in multidimensional databases. Based on the given model of multidimensional database, a set of operations for schema evolution such as inserting/deleting a level in a dimension, adding/deleting an attribute, making an attribute a dimensional level or a fact, etc., are defined by describing the effects of the operations on the databases instances. Using the formal framework as the conceptual model, the goal of this approach is to support the process of automatically propagating the changes from requirements to implementation automatically by transforming the operations to a sequence of SQL commands. However, the work is focused on giving the definition of the operations with no technical details found on the transformation. Furthermore, this approach will not support history-keeping, a property that is desirable in data warehouse.

In [79], the method of describing data warehouse evolution by schema versioning, an area closely related to temporal database, is proposed. The main issue in schema versioning for data warehouse is to keep the historical data and at the same time to support cross version enquiry. This work presents a multiversion data warehouse to support alternative versions of the data warehouse and the data conversions between different versions. By explicitly modelling the version, it supports the cross version and what-if queries. This versioning approach is different from Kimball's as it assigns a version number to a database instead of a dimension. Hence, guidelines on when a version number should be used, particularly when version number is explicitly modelled, are expected.

Comparing with the aforementioned methods, our method differs from them by taking the evolution as one of the design issues. We use schema/view integration based on the notion of schema dominance, which in particular supports history-keeping. Since our conceptual model in HERM does not model the dimensions and facts explicitly, we do not need to define a special set of operations to capture the evolution of our database. We consider a set of materialised views for query performance, so our data warehouse evolution is reflected not only in the schema evolution, but also in the set of materialised views, for which we employ a new view selection algorithm.

2.2 Schema Integration

Database schema integration is an old issue that has attracted a lot of research [57, 60, 63, 67, 106]. The starting point for schema integration is a set of schemata over some data models. Usually the focus is on two schemata over the same data model. If the underlying data models differ, then we may assume some preprocessing transforming both schemata – or one of them, if this is sufficient – into an equivalent schema over a data model with equal/higher expressiveness. Then schema integration aims at replacing the given schemata by a single new one in such a way that the new schema dominates or is equivalent to the old ones.

A view on some database schema consists of another schema called the target schema, and a defining query, which maps instances of the source schema to instances of the target schema. If we integrate the target schemata of views we talk of *view integration* [8, 104, 106]. In this case we obtain embeddings for each target schema into the new schema. If these embeddings are coupled with the defining queries for the given views we obtain a new defining query, i.e. we obtain not only an integrated target schema, but an integrated view.

The work on view integration in [60, 63, 104] is based on the Entity-Relationship model. Larson et al. [63] consider containment, equivalence and overlap relations between attributes and types that are defined by looking at "real world objects". Equivalence between types gives rise to their integration, containment defines a hierarchy with one supertype and one subtype, and overlapping gives rise to a new common supertype. The work by Spaccapietra and Parent [104] considers also relationships, paths and disjointness relations between types. The work by Koh et al. [60] provides additional restructuring rules for the addition or removal of attributes, generalisation and specialisation, and the introduction of surrogate attributes for types.

The work by Biskup and Convent in [8] is based on the relational data model with functional, inclusion and exclusion dependencies. The method is based on the definition of integration conditions, which can be equality, containment, disjointness or selection conditions. Transformations are applied aiming at the elimination of disturbing integration conditions. Just as our work it is based on a solid theory. On the other hand, it has never been applied to large systems in practice. The approach by Sciore et al. in [101] investigates conversion functions on the basis of contexts added to values. These contexts provide properties to enable the semantic comparability of values.

The work by Lehmann and Schewe [66, 67] assumes that the given schemata are defined on the basis of the Higher-order Entity-Relationship model (HERM) [106] which is known to provide enough expressiveness such that schemata existing in practice can be easily represented in HERM. The work relies on the notions of equivalence and dominance as defined for HERM in [106].

In [66, 67] these notions of equivalence and dominance are also compared with those defined by Hull [50] and Qian [86]. Basically, the four different notions of schema dominance introduced by Hull differ by the way the transformation functions are defined. In the simplest case (calculus dominance) they correspond to calculus queries, whereas in the most general case (absolute dominance) there are no restrictions at all. In fact, taking computable functions will remove the arbitrariness from absolute dominance that has been criticised in [86] while taking into account the most general form of queries [24, 113].

In [75], a method for schema (or view) integration following the framework in [67] is presented, i.e. it first “clean” given schemata by removing name conflicts, synonyms and homonyms, then adds inter-schema constraints, and applies to this schema formal equivalence transformation or augmentation rules. The transformation and augmentation rules are correct in the sense that they will always result in a new schema / view that is equivalent to the original one or dominates it. For this a new concept of schema equivalence and dominance based on computable functions is introduced. The work concentrates on finding a reasonable approach to view integration and cooperation that is theoretically founded, but pragmatically oriented.

Our method applies the results from [75] with small modifications in the schema transformation rules, which will be discussed later in Chapter 6.

2.3 Software System Development Methods

Software system development is the process of transforming the user requirements into a computer system. Since the requirements are usually written in a natural language, which is usually imprecise and and ambiguous, and the system is implemented in a computer language which is hard for people to read or understand, this transformation process is rather complex. Therefore, many different methods have been developed. Some are development process oriented, such as the structured system development method, also called waterfall approach, iterative method, and Rational Unified Process (RUP). Others are technique specific, such as prototyping and formal method. The term “formal method” has also been used with a different meaning, namely a method following established standards or a method that is mathematically based. We use the term “formal method” with the latter meaning in this thesis.

2.3.1 Process Oriented Methods - a Brief Introduction

The waterfall approach defines a sequential development process in which development is seen as flowing steadily downwards (like a waterfall) through the phases of requirements analysis, design, implementation, testing (validation), integration, and maintenance [120]. Although the rigidness of the development process is criticised as unsuitable for the iterative nature of the system development process [120], the method has been widely applied. Instead of following the development process strictly as in the waterfall approach, the iterative approach uses a spiral model which works on a set of prototypes before the actual system is built [11]. A prototype is similar to the actual system in feeling but much simpler. It is less costly to construct but can be used for evaluation and further evolution. A new version is constructed by fixing the problems identified in the evaluation and incorporating more user requirements. This process will repeat until either a satisfactory prototype is reached, based on which the final system will be constructed, or a decision is made to give up the idea of the system. The iterative style is particularly suitable when users are not sure what they really want from the system at the very beginning. The RUP method follows an iterative and incremental development process [54]. Each iteration is a process including requirements, design, implementation, testing and deployment, but the focus of the project will shift that defines the four project phases: inception, elaboration, construction and transition. Part of the method is the industry-standard Unified Modelling Language (UML) [120].

2.3.2 Formal Methods

Formal methods are mathematically based techniques and tools for the specification, development and verification of software and hardware systems. Specifically, the mathematical techniques are integrated into the development process through:

- a modelling language with precise syntax and semantics for specification;
- a notion of refinement for stepwise development;
- techniques and tools for verification/validation; and
- a standard methodology for application.

There are many formal methods developed and used, such as the ASM-based method, the B-method, CSP, CCS, π-calculus, LSF, VDM, and Z [119].

The ASM-based method is developed based on Gurevich’s Abstract State Machines (ASMs) by E. Börger for high level system design and analysis [13, 17]. The method is characterized by the imperative style of its modeling language, i.e. the notion of ASMs, which focuses more on the operational semantics than the syntax. Using this method, a computer system is modelled by an ASM, which is a machine whose state consists of functions which can be updated via transition rules. The semantics of the transition rules is defined by interpreting them (in a particular state) as set of updates on the functions (by assignment) in the machine. Each execution of an update set (if consistent) will result in a new state of the machine. A sequence of states related through ASM executions –

finite, when the state of the machine stays the same from somewhere onwards, or infinite – defines a run of the machine. Therefore, the ASM model displays the operational view of a computer system, which makes it easy to understand, but not always easy to verify. However, being executable, ASM modelling enables testing in early stages of development for uncovering errors. Furthermore, the notion of ASMs allows a computer system to be modelled at any given level of abstraction. This forms the foundation of the ASM-based method, which defines the development process from capturing the user requirements by a ground model, at the level of abstraction that closely simulates the user requirements, to further developments by stepwise refinements, reaching a level closer to implementation on a machine. Each refinement realises a single design decision of the system. There are supporting tools developed for system verification and validation [49].

The ASM-based method has been successfully applied in many areas [49], such as formalising database recovery [45], modelling operational semantics of database transactions [84], specification of Java and the Java Virtual Machine (JVM) [18], specification of the MS COM [7], and refinement proving for electronic purse design [95]. Many supporting tools for executing the ASM specifications have been developed [49].

The B-method is a tool-supported formal method based around AMN (Abstract Machine Notation), the programming and specification language for specifying abstract machines in the B-method [1]. It was originally developed by Jean-Raymond Abrial. B is related to the Z notation (also proposed by Abrial) and supports development of programming language code from specifications, rather than just formal specification [118]. The B-method is conceptually close to the ASM method, by using abstract machines, but differs in the way how the abstract machines are constructed, i.e. by constructing the proof at the same time with the support from proof tools [13]. The difference in other areas such as concept, etc. can be found in [13]. The B method has been used in major safety-critical system applications in Europe (such as the Paris Métro Line 14) and other projects [21]. Its tool support for specification, design, proof and code generation is robust and commercially available.

The VDM (Vienna Development Method) method is one of earliest established formal methods for computer-based system development. It originated at IBM's Vienna Laboratory in the 1970s for denotational description of programming languages, and has evolved to include a group of techniques, such as the specification language VDM-SL, data refinement techniques, and operation decomposition techniques [56, 27]. A model in VDM-SL describes a system in terms of the functionality performed on data. It consists of a series of definitions of data types and functions or operations performed upon them. There are many successful applications of VDM in the industry and the supporting tools developed [121].

The Z method focuses on system specification and correctness proof. Its notation, which has been developed at Oxford University's Programming Research Group in the late 1970s and early 1980s [122]. Z notation is based on set theory and mathematical logic. The specification in Z is of a semi-graphic style. The basic building block is called scheme, which is used to describe the states of the system and the operations on the states by modelling the before- and after- states [105, 30]. The Z notation is standardised. It is fully typed which makes it possible to type-check the specifications for uncovering errors in the early design stage. The Z method has evolved to include the notion of data refinement and operation refinement [29]. Successful application and development of Z has been reported in [20].

Besides VDM, Z, B, ASMs as the major methods with industrial impact, there are the algebraic specifications with OBJ, Maude, etc., and the process calculi such as CSP, CCS, π-calculus, LSF, etc.

2.3.3 Application of Formal methods

As pointed out in [47, 23], the major benefit of formal methods is the abstract and precise system specification, which helps in controlling the system complexity, removing ambiguity and inconsistency in system requirements, and facilitating system correctness proof. Furthermore, errors uncovered in the early stage of system development reduce the system cost. Formal methods are useful in any system development, and are particularly desired in systems being critical in terms of safety, security or information, such as aviation, networking, and management decision support.

However, formal methods are not accepted as common practice due to the perception of being hard to understand and apply [47, 23], and due to the extra work required. Much effort has been made in making the application of formal methods a less difficult and more beneficial process, e.g. by providing supporting tools for specification generation (e.g. CASE tools), validation or testing, system aided verification, etc., by tailoring the method toward the application areas as shown in [97, 100], and by integrating the techniques of the formal methods with the other software development practice as shown in [22, 46, 3, 85, 114]).

In [97], a B-like language is used for modelling programs in database systems. It presents a set of refinement rules for correct program development by construction. The definition of refinement rules follows the style of weakest precondition.

In [100], we proposed a typed ASM for modelling database schema in HERM (Higher-order Entity-Relationship Model). By incorporating the types into the ASM, a more specific notion of refinement, i.e. strong schema refinement, is built for the notion of schema dominance. The notion of strong schema refinement is used as the base for a set of refinement rules built for data warehouse schema evolution, which play an important role for ensuring integrity in the development of data warehouses. There are other approaches to introducing typed versions of ASMs, e.g. [28] following similar ideas, but with different focus, which lead to using different type systems. The work in [124] presents a very general approach to combine ASMs with type theory.

In [22], instead of a using fully formal method or informal method, a method that integrates the user-centred method for interface design with a formal environment is presented, aiming to realise the benefits of both methods.

2.4 Summary

The review of literature in this chapter has covered the research areas of design methods and techniques in data warehousing, integration techniques in database design, and the formal methods in system development. In the data warehousing area, it has shown that the work for an integrated design method is still in the preliminary stage, in particular, the dynamic issue is not well addressed. We consider businesses to be of a dynamic nature, and consequently management decision support systems are as well. Hence, it is important to address the dynamic issues in the design method. In schema integration, matured techniques are available and are applied in data intensive system successfully, but not yet in data warehousing. As data consistency and integrity is critical for the success of management support systems, it is a serious concern in data warehouse design. We believe that consistency and integrity can be improved by incorporating the schema integration technique in the data warehouse design, especially in the “bottom-up” approach. The review in the formal methods area has shown that there are many successful formal methods.

Our goal is not to compare different formal methods, several of which may be equally suitable for our task. Instead we focus on the rigidity that a formal methods can offer in the development process. Specifically, we use it to manage the complexity of the system, by abstraction, stepwise refinements and precise documentation of design steps for future changes. Furthermore, the review has also shown that in order to make it easy to integrate the formal method into the current system development process for a particular domain, adaptation and specific pragmatics are useful.

Chapter 3

Abstract State Machines

Abstract State Machines (ASMs)(formerly known as the Evolving Algebras, [42, 43]) were created by Yuri Gurevich as an attempt to bridge the gap between formal models of computation and practical specification methods by improving on the Turing's thesis. That is, according to Gurevich [43], to seek "more versatile machines which would be able to simulate arbitrary algorithms in a direct and essentially code-free way. ... The simulator is not supposed to implement the algorithm on a lower level; the simulator should be performed on the natural abstract level of the algorithm.". Here "the natural level" is used to contrast the level that Turing machines use to simulate algorithms.

The ASM thesis presents a way to model the real world so closely that the correctness can be established by observation or testing. The model can then be refined or coarsened for many purposes [43].

Since the establishment, ASMs have been widely used in specifying languages, real and virtual architectures, in validating language implementations and distributed protocols, and in proving complexity result, etc. [49]. In particular, Egon Börger has found its usefulness in system development and developed it into a method for high-level system design and analysis [17]. The general idea of the ASM approach is to provide a formal framework for the entire system development process, i.e. starting from the requirements capturing, through the stepwise refinement, leading to the implementation with a simple and intuitive mathematical form and without dropping into the pitfall of the "formal methods straight-jacket".

The three constituents in the ASM method are: the notion of ASMs for capturing the fundamental operational concepts of computing; the ground model method for capturing requirements; and the refinement method for turning the ground model by incremental steps into implementation.

For the purpose of being consistent, free of confusion and self-contained, a complete set of definitions for the ASM method are taken from [18, 17, 43] and shown in the following sections. In addition, this chapter also includes discussions on refinement correctness proof and a comparison on refinement with other formal methods.

3.1 The Notion of ASMs

Basic ASMs are finite sets of *transition rules* of the form

if *Condition* **then** *Updates*

which transform abstract states. The *Condition* (also called *guard*) is an arbitrary predicate logic formula without free variables. The *Updates* rule is fired iff its guard evaluates to true. *Updates* is a finite set of assignments of the form

$$f(a_1, \ldots, a_n) := v$$

whose execution changes (or defines if previously undefined) in parallel the value of the occurring functions f at the indicated arguments $a_1, \ldots, a_n$ to the indicated value v.

The notion of ASM *states* is the classical notion of mathematical *structures* where data comes as abstract objects, i.e., as elements of sets (domains, *universes*, one for each category of data) which are equipped with basic operations(partial *functions*) and predicates (attributes or relations). By default, it includes equality sign, the nullary operations *true*, *false*, and *undef* and the boolean operations.

The notion of ASM *run* is the classical notion of computation of transition systems. An ASM computation step in a given state consists in executing simultaneously all updates of all transition rules whose guard is true in the state, if these updates are consistent. For the evaluation of terms and formulae in an ASM state, the standard interpretation of function symbols by the corresponding functions in that state is used.

Functions are classified as *basic* or *derived* functions. *Basic* functions are functions which are part of the state, while *derived* functions are a kind of auxiliary functions, which may vary over time but are not updatable directly by neither the ASM nor the environment. *Basic* functions are further classified into *static* or *dynamic* functions. *Static* functions are functions which never change while *dynamic* functions may change during the run of the ASM. *Dynamic* functions can be further divided into four subclasses, depending on who is allowed to update them:

- either by and only by the ASM, in which case we get *controlled* functions,
- by the environment, in which case we get *monitored* functions,
- by both, then we get *shared* functions,
- updatable but not readable by the ASM and readable only by environment, which gives us *out* functions.

In particular, a dynamic function of arity 0 acts as a variable, whereas a static function of arity 0 acts as a constant.

3.2 Mathematical Definition of ASMs

This section provides a mathematical definition for the syntax and semantics of ASMs.

3.2.1 Abstract States

In an ASM state, data comes as abstract elements of domains (also called universes) which are equipped with basic operations. The states of ASMs "store" the current value of basic functions. As they form *algebras*, as introduced in standard logic or universal algebra textbooks, and may change over time, ASMs where initially called "evolving algebras".

Definition 3.1 (Signature)**.** [17, Def 2.4.1] A *signature* Σ is a finite collection of function names. Each function name f has an *arity*, a non-negative integer. The arity of a function name is the number of arguments the function takes. Function names can be *static* or *dynamic*. Static nullary function names are called *constants*, dynamic nullary functions correspond to the variables of programming. Every ASM signature is assumed to contain constant *undef, True, False*. Signatures are also called *vocabularies*.

Definition 3.2 (State)**.** [17, Def 2.4.2] A *state* $\mathfrak{A}$ of the signature Σ is a non-empty set X, the *superuniverse* of $\mathfrak{A}$, together with *interpretations* of the function names of Σ. If f is an n-ary function name of Σ, then its interpretation $f^{\mathfrak{A}}$ is a function from X^n into X; if c is a constant of Σ, then its interpretation $c^{\mathfrak{A}}$ is an element of X. The superuniverse X of the state $\mathfrak{A}$ is denoted by $|\mathfrak{A}|$.

Formally, function names are interpreted in states as total functions. They can however be viewed as being partial, if we define the *domain* of an n-ary function name f in $\mathfrak{A}$ to be the set of all n-tuples $(a_1, \ldots, a_n) \in |\mathfrak{A}|^n$ such that $f^{\mathfrak{A}}(a_1, \ldots, a_n) \neq \mathit{undef}^{\mathfrak{A}}$. The constant *undef* represents an undetermined object, the default value of the superuniverse. In applications, the superuniverse X of a state $\mathfrak{A}$ is usually divided into smaller *universes*, modelled by their characteristic functions.

Definition 3.3 (Term)**.** [17, Def 2.4.11] The terms of $\mathfrak{A}$ are syntactic expressions generated as follows:

- Variable $v_0, v_1, \ldots$ are terms.
- Constants c of $\mathfrak{A}$ are terms.
- If f is an n-ary function name of Σ and $t_1, \ldots, t_n$ are terms, then $f(t_1, \ldots, t_n)$ is a term.

A term which does not contain variables is called *closed.* Note that term variables are *not* dynamic functions of arity 0. Terms are purely syntactic objects, but can be interpreted in a state when given a variable assignment (see below).

Definition 3.4 (Variable assignment)**.** [17, Def 2.4.12] Let $\mathfrak{A}$ be a state. A *variable assignment* for $\mathfrak{A}$ is a function ζ which assigns to each variable v_i an element $\zeta(v_i) \in |\mathfrak{A}|$. $\zeta\frac{a}{x}$ is used for the variable assignment which coincides with ζ except that it assigns the element a to the variable x. So we have:

$$\zeta\frac{a}{x}(v_i) = \begin{cases} a & \text{if } v_i = x; \\ \zeta(v_i) & \text{otherwise.} \end{cases}$$

Definition 3.5 (Interpretation of terms)**.** [17, Def 2.4.13] Let $\mathfrak{A}$ be a state of Σ, ζ be a variable assignment for $\mathfrak{A}$ and t be a term of Σ. By induction on the length of t, a value $[\![t]\!]^{\mathfrak{A}}_{\zeta}$ is defined as follows:

- $[\![v_i]\!]^{\mathfrak{A}}_{\zeta} := \zeta(v_i)$,
- $[\![c]\!]^{\mathfrak{A}}_{\zeta} := c^{\mathfrak{A}}$,
- $[\![f(t_1, \ldots, t_n)]\!]^{\mathfrak{A}}_{\zeta} := f^{\mathfrak{A}}([\![t_1]\!]^{\mathfrak{A}}_{\zeta}, \ldots, [\![t_1]\!]^{\mathfrak{A}}_{\zeta})$.

The interpretation of t depends on the values of ζ on the variables of t only: if $\zeta(x) = \xi(x)$ for all variables x of t, then $[\![t]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\xi}$.

Definition 3.6 (Boolean Formula)**.** [17, Def 2.4.14] Let Σ be a signature. The formulas of Σ are generated as follows:

- If s and t are terms of Σ, then $s = t$ is a formula.
- If φ is a formula, then $\neg\varphi$ is a formula.

- If φ and ψ are formulas, then $(\varphi \land \psi)$, $(\varphi \lor \psi)$ and $(\varphi \to \psi)$ are formulas.
- If φ is a formula and x a variable, then $(\forall x \varphi)$ and $(\exists x \varphi)$ are formulas.

The logical connectives and quantifiers have the standard meaning. e.g. $\to$ is called implication, and it has the meaning if-then.

A formula $s = t$ is called an *equation*. The expression $s \neq t$ is an abbreviation for the formula $\neg(s = t)$. Parentheses in formulas are often omitted for readability where it does not create ambiguities. Formulas can be interpreted in a state with respect to a variable assignment. The classical truth tables for the logical connectives and the classical interpretation of quantifiers are used. The equality sign is interpreted as identity.

Definition 3.7 (Interpretation of formulas)**.** [17, Def 2.4.15] Let $\mathfrak{A}$ be a state of Σ, φ be a formula of Σ and ζ be a variable assignment in $\mathfrak{A}$. By induction on the length of φ, a truth value $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} \in \{\mathit{True}, \mathit{False}\}$ is defined as follows:

$$
\begin{aligned}
[\![s = t]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![s]\!]^{\mathfrak{A}}_{\zeta} = [\![t]\!]^{\mathfrak{A}}_{\zeta}; \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\neg\varphi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False}; \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\varphi \land \psi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True} \text{ and } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}; \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\varphi \lor \psi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True} \text{ or } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}; \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\varphi \to \psi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\psi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True} \text{ or } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False}; \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\forall x \varphi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True} \text{ for all } a \in |\mathfrak{A}| \text{ ;} \\ \mathit{False} & \text{otherwise.} \end{cases} \\
[\![\exists x \varphi]\!]^{\mathfrak{A}}_{\zeta} &:= \begin{cases} \mathit{True} & \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True} \text{ for some } a \in |\mathfrak{A}| \text{ ;} \\ \mathit{False} & \text{otherwise.} \end{cases}
\end{aligned}
$$

A state $\mathfrak{A}$ is called a *model* of φ, if $[\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}$ for all variable assignments ζ.

3.2.2 Transition Rules and Runs

Updating states means to change the interpretation of (some of) the functions in the underlying signature. The way ASMs update states is described by transition rules of the following form which define the syntax of ASM programs.

Note that this notion of ASMs is different from basic ASMs: instead of having sets of updates using terms, we have transition rules, which first need to be interpreted in a particular state to result in an update set.

Definition 3.8 (Transition rules)**.** [17, p72] Let Σ be a signature. The transition rules R, S of an ASM are syntactic expressions generated as follows:

1. *Skip Rule:*

   skip

   Meaning: do nothing.

2. *Update Rule:*

   $f(t_1, \ldots, t_n) := s$

   Syntactic conditions:

   - f is an n-ary, dynamic function name of Σ
   - $t_1, \ldots, t_n$ and s are terms of Σ

   Meaning: in the next state, the value of the function f at the arguments $t_1, \ldots, t_n$ is updated to s.

3. *Block Rule:*

   RS

   Meaning: R and S are executed in parallel.
   Note: We will also write $R||S$ where this is more readable.

4. *Sequential Rule*

   $R_1; \ldots ; R_n$

   Meaning: the rules $R_1, \ldots, R_n$ will be executed sequentially.

5. *Conditional Rule:*

   `if` φ `then` R `else` S

   Meaning: if φ is true, then execute R, otherwise execute S.

   This rule can be extended to:

   `if` φ_1 `then` R_1 `elsif` $\ldots$ `elsif` φ_n `then` R_n

   The meaning is obvious.

   For readability, we introduce a short form for the above rule by *Case Rule*, if φ_i is the expression $v = c_i$:

   `case` v `of` $c_1 : R_1, \ldots, c_n : R_n$ `endcase`

6. *Let Rule*

   `let` $x = t$ `in` R

   Meaning: assign the value of t to x and execute R.

7. *Forall Rule:*

   `forall` x `with` φ `do` R

   Meaning: execute R in parallel for each x satisfying φ.

8. *Choose Rule:*

   `choose` x `with` φ `do` R

   Meaning: choose an x satisfying φ and then execute R.

9. *Call Rule:*

   $r(t_1, \dots, t_n)$

   Meaning: call r with parameters $t_1, \dots, t_n$.

Definition 3.9 (Rule declaration)**.** [17, Def 2.4.18] A *rule declaration* for a rule name r of arity n is an expression of the form

$$r(x_1, \dots, x_n) = R$$

where R is a transition rule and the free variables of R are contained in the list $x_1, \dots, x_n$. In a rule call $r(t_1, \dots, t_n)$ the variables x_i in the body of R of the rule declaration are replaced by the values of the parameters t_i (call by value).

Definition 3.10 (ASM)**.** [17, Def 2.4.19] An *abstract state machine* M consists of a signature Σ, an initial state $\mathfrak{A}$ for Σ, a rule definition for each rule name, and a distinguished rule name of arity zero called the *main rule name* of the machine.

Definition 3.11 (Update)**.** [17, Def 2.4.4] An *update* for $\mathfrak{A}$ is a triple $(f, (a_1, \dots, a_n), b)$, where f is an n-ary dynamic function name, and $a_1, \dots, a_n$ and b are elements of $|\mathfrak{A}|$.

The meaning of an update $(f, (a_1, \dots, a_n), b)$ is that the interpretation of the function f in $\mathfrak{A}$ has to be changed at the arguments $a_1, \dots, a_n$ to the value b. The pair of the first two components of an update is called a *location*. An *update set* is a set of updates.

Definition 3.12 (Consistent update set)**.** [17, Def 2.4.5] An update set U is called *consistent*, if it satisfies the following property:

$$\text{if } (f, (a_1, \dots, a_n), b) \in U \text{ and } (f, (a_1, \dots, a_n), c) \in U, \text{ then } b = c$$

Definition 3.13 (Firing of updates)**.** [17, Def 2.4.6] The result of firing a consistent update set U in a state $\mathfrak{A}$ is a new state $\mathfrak{B} = \mathfrak{A} + U$ with the same superuniverse as $\mathfrak{A}$ satisfying the following two conditions for the interpretations of function names f of Σ:

1. If $(f, (a_1, \dots, a_n), b) \in U$, then $f^{\mathfrak{B}}(a_1, \dots, a_n) = b$;

2. If there is no b with $(f, (a_1, \dots, a_n), b) \in U$ and f is not a monitored function, then $f^{\mathfrak{B}}(a_1, \dots, a_n) = f^{\mathfrak{A}}(a_1, \dots, a_n)$

Definition 3.14 (Composition of update sets)**.** [17, Def 2.4.10]

$$U \oplus V = V \cup \{(l, v) \in U \mid \text{ there is no } w \text{ with } (l, w) \in V\}$$

Applying the update set $U \oplus V$ to a state $\mathfrak{A}$ is the same as first applying U and then applying V to the resulting state $\mathfrak{A} + U$.

Given a state and variable assignment, a transition rule of an ASM produces an update set. A definition for the semantics of a transition rule via a calculus is given next. Note that since we allow recursive calls, it is possible that a transition rule does not have valid semantics. This happens if the recursive call does not terminate.

$$\frac{}{[\![\texttt{skip}]\!]^{\mathfrak{A}}_{\zeta} \rhd \emptyset}$$

$$\frac{}{[\![f(t) := s]\!]^{\mathfrak{A}}_{\zeta} \rhd \{(f,a,b)\}} \quad \text{if } a = [\![t]\!]^{\mathfrak{A}}_{\zeta} \text{ and } b = [\![s]\!]^{\mathfrak{A}}_{\zeta}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \rhd U \quad [\![S]\!]^{\mathfrak{A}}_{\zeta} \rhd V}{[\![R\ S]\!]^{\mathfrak{A}}_{\zeta} \rhd U \cup V}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \rhd U [\![S]\!]^{\mathfrak{A}+U}_{\zeta} \rhd V}{[\![R; S]\!]^{\mathfrak{A}}_{\zeta} \rhd U \oplus V} \quad \text{if } U \text{ is consistent.}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \rhd U}{[\![R; S]\!]^{\mathfrak{A}}_{\zeta} \rhd U} \quad \text{if } U \text{ is inconsistent.}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta} \rhd U}{[\![\texttt{if } \varphi \texttt{ then } R \texttt{ else } S]\!]^{\mathfrak{A}}_{\zeta} \rhd U} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}$$

$$\frac{[\![R_i]\!]^{\mathfrak{A}}_{\zeta} \rhd U_i}{\left[\!\!\left[\begin{array}{l}\texttt{if } \varphi_1 \texttt{ then } R_1 \\ \texttt{elsif } \varphi_2 \texttt{ then } R_2 \\ \texttt{elsif } \ldots\ R_n\end{array}\right]\!\!\right]^{\mathfrak{A}}_{\zeta} \rhd U_i} \quad \text{if } [\![\varphi_j]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False} \text{ for } j < i \text{ and } [\![\varphi_i]\!]^{\mathfrak{A}}_{\zeta} = \mathit{True}$$

$$\frac{[\![S]\!]^{\mathfrak{A}}_{\zeta} \rhd U}{[\![\texttt{if } \varphi \texttt{ then } R \texttt{ else } S]\!]^{\mathfrak{A}}_{\zeta} \rhd U} \quad \text{if } [\![\varphi]\!]^{\mathfrak{A}}_{\zeta} = \mathit{False}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \rhd U}{[\![\texttt{let } x = t \texttt{ in } R]\!]^{\mathfrak{A}}_{\zeta} \rhd U} \quad \text{if } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \rhd U_a \text{ for each } a \in I}{[\![\ \texttt{forall } x \texttt{ with } \varphi \texttt{ do } R]\!]^{\mathfrak{A}}_{\zeta} \rhd \bigcup_{a\in I} U_a} \quad \text{if } I = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True}\}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \rhd U_a \text{ for some } a \in I}{[\![\ \texttt{choose } x \texttt{ with } \varphi \texttt{ do } R]\!]^{\mathfrak{A}}_{\zeta} \rhd U_a} \quad \text{if } I = \{a \in |\mathfrak{A}| : [\![\varphi]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} = \mathit{True}\}$$

$$\frac{[\![R]\!]^{\mathfrak{A}}_{\zeta\frac{a}{x}} \rhd U}{[\![r(t)]\!]^{\mathfrak{A}}_{\zeta} \rhd U} \quad \text{if } r(x) = R \text{ is a rule definition and } a = [\![t]\!]^{\mathfrak{A}}_{\zeta}$$

Definition 3.15 (Semantics of transition rules)**.** [17, Def 2.4.20] The semantics of a transition rule R of a given ASM in a state $\mathfrak{A}$ with respect to a variable assignment ζ is defined if and only if there exists an update set U such that $[\![R]\!]_\zeta^{\mathfrak{A}} \triangleright U$ can be derived in the given calculus.

Definition 3.16 (Move of an ASM)**.** [17, Def 2.4.21] Let ζ be a variable assignment. We say that a machine M can make a move from state $\mathfrak{A}$ to $\mathfrak{B}$ (written $\mathfrak{A} \Rightarrow \mathfrak{B}$), if the main rule of r yields a consistent update set U in state $\mathfrak{A}$ using ζ and $\mathfrak{B} = \mathfrak{A} + U$.

Definition 3.17 (Run of an ASM)**.** [17, Def 2.4.22] Let M be an ASM with signature Σ. A *run* of M is a finite or infinite sequence $\mathfrak{A}_0, \mathfrak{A}_1, \ldots$ of states for Σ such that $\mathfrak{A}_0$ is an initial state of M and for each n, either M can make a move from $\mathfrak{A}_n$ into the next internal state $\mathfrak{A}'_n$ and the environment produces a consistent set of external or shared updates U such that $\mathfrak{A}_{n+1} = \mathfrak{A}'_n + U$, or M cannot make a move in state $\mathfrak{A}_n$ and $\mathfrak{A}_n$ is the last state in the run.

By allowing the environment to contribute to the update set, we can model communication with external processes, such as e.g. getting input over time (instead of making all input part of the initial state).

3.2.3 The Reserve of ASMs

To introduce new, previously unused values, a special universe called reserve is used. The ASM reserve set is part of the Universe. New elements are allocated using the rule

`import` x `do` R

Meaning: choose an element x from the reserve, delete it from the reserve and execute R.

The reserve of a state is special unary, dynamic relation *Reserve* which can not be updated directly by an ASM using the update rule, but will be updated automatically when an `import` statement is executed.

3.3 ASM Modules

For complex and large ASMs, a standard module concept can be applied. This does not change the underlying semantics, but allows for better code structuring. We construct a large ASM by a collection of ASM modules (also called submachine) $M_1, \ldots, M_n$. Each M_i consists of a *header* and a *body*. The header of an ASM consists of its name, an import- and export-interface, and a signature. The body of an ASM module consists of function declarations and rule definitions. Thus, an ASM module is written in the form

```
ASM $M$
IMPORT $M_1(r_{11}, \ldots, r_{1n_1}), \ldots, M_k(r_{k1}, \ldots, r_{kn_k})$
EXPORT $q_1, \ldots, q_\ell$
SIGNATURE $s$
BODY $decl_1 \ldots decl_n$
```

Here r_{ij} are the names for functions or rules which are imported from the ASM M_i. These functions and rules will be declared and defined in the body of M_i — not in the body of M — and only used in M. This is only possible for those functions and rules that have explicitly been exported. So only the functions and rules $q_1, \ldots, q_\ell$ listed after the

EXPORT can be imported and used by other ASMs. For each ASM module there must be a main rule defined. As in standard modular programming languages this mechanism of import- and export-interface permits ASMs to be developed rather independently from each other, leaving the definition of particular functions and rules to "elsewhere".

3.4 Distributed ASMs

The notion of ASMs which formalise simultaneous parallel actions of a single agent has grown into a generalisation where multiple agents act and interact in an asynchronous manner. The following definitions are taken from [43].

Definition 3.18 (Distributed ASM)**.** A distributed ASM M consists of the following:

1. A finite indexed set of single-agent programs π_ν, called *modules*. The *module names* ν are static nullary function names.
2. A signature $\Sigma = Fun(M)$ ($Fun(M)$ denotes the signature of M) which includes each $Fun(\pi_\nu) - \{\text{Self}\}$ but does not contain Self. Self is a special unary function interpreted differently by different agent, e.g., an agent a interprets Self as a. Thus function Self allows an agent to identify itself among other agents. Self is a logic name and cannot be the subject of an update instruction.
3. Σ contains a unary function name Mod.
4. A collection of Σ-states, called *initial states* of M, satisfying the following conditions:
    - Different module names are interpreted as different elements (agents).
    - There are only finitely many elements a such that, for some module name ν, $\text{Mod}(a) = \nu$.

The requirements for initial states above also apply to non-initial states $\mathfrak{A}$ of M. The *agents* of state $\mathfrak{A}$ are those elements a for which there exists a module ν such that $\mathfrak{A} \models \text{Mod}(a) = \nu$. The corresponding π_ν is the program $\text{Prog}(a)$ of a, and $\text{Fun}(\pi_\nu)$ is the signature $\text{Fun}(a)$ of a. Agent a is deterministic if $\text{Prog}(a)$ is. The underlying idea is to code a module once, and then run multiple copies of it if desired.

$\text{View}_a(\mathfrak{A})$ is the reduction of $\mathfrak{A}$ to signature $\text{Fun}(a) - \{\text{Self}\}$ expanded with Self, which is interpreted as a. $\text{View}_a(\mathfrak{A})$ can be seen as the local state of agent a corresponding to the global state $\mathfrak{A}$.

An agent can make a *move* at $\mathfrak{A}$ by firing $\text{Prog}(a)$ at $\text{View}_a(\mathfrak{A})$ and changing $\mathfrak{A}$ accordingly. To perform a move of an agent a, fire

$$\text{Updates}(a, \mathfrak{A}) = \text{Updates}(\text{Prog}(a), \text{View}_a(\mathfrak{A})).$$

Definition 3.19 (Sequential runs)**.** A *pure sequential run* ρ of an ASM M is a sequence $(S_n : n < k)$ of states of M, where S_0 is an initial state and every S_{n+1} is obtained from S_n by executing a move of an agent.

An *initial segment* of a poset (partially ordered set) P is a substructure X of P which is downward closed, i.e., if $x \in X$ and $y < x$ in P then $y \in X$. As a substructure, X inherits the ordering of P. A *linearisation* of a poset P extends the partial order of P to a linear order, making every two elements in P comparable.

Definition 3.20 (Partially ordered runs)**.** For simplicity, attention is restricted to pure runs and deterministic agents. A *run* ρ of a distributed ASM M can be defined as a triple (P, A, σ) satisfying the following conditions:

1. P is a partially ordered set, where all sets $\{y : y \leq x\}$ are finite.

   Elements of P represent *moves* made by various agents during the run. If $y < x$ then x starts when y is already finished, which is why the set $\{y : y \leq x\}$ is finite.

2. A is a function on P such that every nonempty set $\{x : A(x) = a\}$ is linearly ordered.

   $A(x)$ is the agent performing move x. The moves of any single agent are supposed to be linearly ordered.

3. σ assigns a state of M to the empty set and each finite initial segment of P; $\sigma(\emptyset)$ is an initial state.

   $\sigma(X)$ is the result of performing all moves in X.

4. The coherence condition: if x is a maximal element in a finite initial segment X of P and $Y = X - x$, then $A(x)$ is an agent in $\sigma(Y)$ and $\sigma(X)$ is obtained from $\sigma(Y)$ by firing $A(x)$ at $\sigma(Y)$.

In particular, the conditions above ensures that every linearisation of a finite initial segment of P leads to the same final state.

Partially ordered runs can be used to restrict the order of moves among ASMs when there are cooperative actions defined. Consider e.g. a setup where ASM M_1 sends a value to a shared location l, and ASM M_2 retrieves the value from l. The moves in the two ASMs, M_1 and M_2, which have the sending and retrieving actions, must be ordered, while the remaining moves need not be restricted.

3.5 The Ground Model Method

The ground model method deals with the issues at the beginning of the development process, i.e. user requirements capturing. The discussion of the ground model method in the following is based on [14].

Usually, the first system specification, which is formulated in ASMs straight from the user requirements in natural language, is called the ground model in the ASM method. In the case of reverse engineering, the ground model is the concrete one from which more abstract models are derived by abstraction.

3.5.1 The Properties of the Ground Model

The fundamental problem in building computer systems is deciding precisely what to build. The user requirements are to describe what to build, but often their formulation is incomplete or too detailed, ambiguous or inconsistent. Hence, we need a model, namely a ground model, for capturing the requirements, to be

- *precise* at the appropriate level of detailing yet *flexible*, to satisfy the required accuracy exactly, without adding unnecessary precision;
- *simple and concise* to be understandable by both domain experts and system designer, and to be manageable for inspection and analysis - this is achieved by "directly" reflecting the structure of the real-world problem;

- *abstract (minimal) yet complete.* Completeness means that every semantically relevant features is present with no hidden clauses. The completeness property requires the designer to include a statement of the assumptions made for them at the abstract level and to be realized through the detailed specification left for the later refinements.
- *validatable* and thus in principle falsifiable by experiment.
- equipped with simple yet *precise semantical foundation* as a prerequisite for rigorous analysis and as a basis for reliable tool development and prototyping.

3.5.2 Three Problems in Formulation

The difficult and error-prone task in requirements capturing is the formalisation task, that is, to translate the usually natural-language problem description into a sufficiently precise, unambiguous, consistent, complete (but different from program code), and minimal formulation of the“conceptual construct” of a computer system.

There are three essential problems in the requirements formalisation, namely

- the *language and communication problem*,
- the *verification-method problem*, and
- the *validation problem*.

The language and communication problem happens between the domain expert and the system designer in deciding what to build. As the common understanding of ”what to build” will be used as a contract to bind the two parties in the rest of the development process, it is to be documented in such a model that both the domain expert and the system designer are able to understand. This requires the language to be suitable to model the problem closely (at the right level of abstraction) focusing on the domain issues of the given problem.

The verification problem stems from the fact that there are no mathematical means to prove the correctness of the passage from an informal to a precise description. This can only be replaced by inspections on some kind of “evidence” of the desired correspondence between the informal model and the precisely formalised model, namely the ground model. This requires that the ground model inspection provides the evidence of correctness. To establish the system completeness and consistency, it requires that the ground model can be used to describe the original intention of the system, and to express it correctly to the designer, and it can also be used by the domain expert to inspect its completeness, as well as the designer to check formally the internal consistency and the consistency of different system views.

The validation problem is about the possibility of performing experiments with the ground model, in particular to simulate it for running relevant scenarios (use cases), providing a framework for systematic attempts to falsify the model against the counter part in reality.

3.5.3 ASMs for the Formalisation Problems

ASMs solve the language and communication problem due to the notion of ASMs, which is easy to understand for both system designers, by the imperative style, and domain experts, by problem-orientation, and allows one to tailor the ground model to resemble the structure of the real-world problem.

Using ASMs as ground models eases the verification problem since it allows one to use both inspection (through easy to understand) and reasoning (mathematical based) with other means that are appropriate. The notation does not limit the verification space.

The validation problem is solved by the operational semantics of ground model ASMs (by executable ground model or program walk through), which come with a standard notion of computation or“run”.

3.5.4 An Example of Ground Model

A small example, simplified from Börger’s work [17, pp. 89-91] is used in the following to show how an ASM ground model is derived from user requirements.

Example 3.1. The problem description of an order invoicing system:

R0.1 The subject is to invoice orders.

R0.2 To invoice is to change the state of an order, from *pending* to *invoiced.*

R0.3 On an order, we have one and only one reference to an ordered product of a certain quantity. The quantity can be different from other orders.

R0.4 The same reference can be ordered on several different orders.

R0.5 The state of the order will be changed to invoiced if the ordered quantity is either less than or equal to the quantity which is in stock according to the reference of the ordered product.

R1.1 All the ordered references are in stock.

R1.2 The stock or the set of the orders may vary due to the entry of new orders or cancelation of orders, or due to having a new entry of quantities of products in stock at the warehouse. But we do not have to take these entries into account.

R1.3 This means that you will not receive two entry flows, orders or entries in stock. The stock and the set of orders are always given to you in an up-to-date state.

From the above requirements, we can derive the state of the system as follows:

- By R0.1 and R1.3, we have a universe *ORDER*, with neither initialisation nor bound specified.
- By R0.2 there is a dynamic function *state*:*ORDER* → *pending,invoiced.*
- By R0.1 and R0.2, it implies that initially *state(o)=pending* for all the orders *o*.
- By R0.5 and R1.1, we have two universes *PRODUCT* and *QUANTITY*.
- By R0.3 and R1.1, we have two dynamic functions, *product*:*ORDER* → *PRODUCT*, and *orderQuantity*:*ORDER* → *QUANTITY*.
- By R0.5, there is a dynamic function *stockQuantity*:*PRODUCT* → *QUANTITY*, but it does not say when and by who it should be updated.

The dynamic part of the system can be derived from R0.1 and R0.2: one transition updates the state of orders from pending to invoiced. Due to limited stock, it may happen that only a subset of all orders for the same product can be invoiced. Assuming we update one order at a time and proceed as long as condition R0.5 is met, the rule can be formalised as follows:

```
SINGLEORDER =
choose $Order \in ORDER$ with $state(Order) = pending$
   $\wedge\ orderQuantity(\text{Order}) \leq stockQuantity(product(Order))$
   do $state(Order) := invoiced$
      DELETESTOCK($orderQuantity(Order)$, $product(Order)$)
```

□

3.6 The ASM Refinement Method

The ASM refinement method is introduced mainly by referring the works in [15, 93, 92]. The refinement correctness proof is discussed in 3.6.3, and a comparison on refinement notion is given in 3.6.4.

In the ASM method, stepwise refinement is used as a practical method to build a system from ground models, through well-documented incremental design steps, into executable code (or an abstract model in case of reverse engineering). The ASM method is problem oriented and geared to support divide-and-conquer techniques for both design and verification. This is similar to the approach taken in [123, 26] for programming by abstraction, structuring, decomposition and verification. The ASM method provides the freedom of abstraction, i.e. to allow arbitrary structures to reflect the underlying notion of state, and the notion of refinement to map an abstract machine to a more concrete machine with its observable states and runs in such a way that the desired equivalence can be established under well defined conditions.

3.6.1 The Notion of Refinement

The general notion of *ASM refinement* between two ASMs M and M^* is built up using

- a correspondence between the states s of M and the states s^* of M^*, and
- a correspondence between the runs of M and M^* involving states s and s^*, respectively.

Definition 3.21 (Correct Refinement). Fix any notion $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M iff for each M^*-run $S_0^*, S_1^*, \dots$ there is an M-run $S_0, S_1, \dots$ and sequences $i_0 < i_1 < \dots, j_0 < j_1 < \dots$ such that $i_0 = j_0 = 0$, and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \dots, j_0 < j_1 < \dots$ are infinite.

We say that the M^*-run $S_0^*, S_1^*, \dots$ simulates the M-run $S_0, S_1, \dots$. The states S_{i_k}, S_{j_k} related by the notion of $\equiv$ are called the corresponding states of interest, which are the states that are observably equal in the corresponding runs.

$$\begin{tikzcd}
\text{State } S \arrow[rrr, "t_1 \;\ldots\; t_m \quad \text{m steps of } M"] \arrow[d, leftrightarrow, "\equiv"'] & & & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[rrr, "u_1 \;\ldots\; u_n \quad \text{n steps of } M^*"'] & & & S^{*\prime}
\end{tikzcd}$$

Figure 3.1: The ASM refinement scheme

According to the general notion of refinement, to relate two ASMs, an ASM M^* refining an ASM M, the designer has the freedom to define the following items:

- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^*, usually constructed using
  - a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of locations to relate in corresponding states, and
  - a notion of *equivalence* $\equiv$ of the data in the locations of interest
- a notion of abstract *computation segments* which consists of m steps $\tau_1, \ldots, \tau_m$ of M, and of corresponding refined computation segments, n steps $\sigma_1, \ldots, \sigma_n$ of M^*. The corresponding computation segments start from corresponding states of interest and end in the next corresponding states of interest, as shown in the general ASM refinement scheme, Figure 3.1. While this is not needed do characterize refinement, it is often vital for constructing a refinement proof.

The focus of the ASM refinement method, as Börger pointed out in [15], is to support the usage of refinements to correctly reflect and explicitly document an intended design decision, adding more details to a more abstract design description, e.g. for making an abstract program executable, for improving a program by additional features or by restricting it through precise boundary conditions which exclude certain undesired behaviours.

3.6.2 The Refinement Patterns

There are three frequently used types of ASM refinement pattern which include the conservative extension, procedural refinement, and data refinement.

Conservative extension is a purely incremental refinement which is used to introduce new behaviour in an existing machine. This is done by first defining the condition for the new behaviour such that it is exclusive from the conditions defined in the existing machine, and then defining the new behaviour, and making sure the existing machine performs the same as before by restricting it using the negation of the new condition. In particular, the conservative extension is used when change of user requirements is purely incremental.

Procedural refinement, also called submachine refinement, consists of replacing one submachine by another submachine in an existing machine. This is often used in restructuring the machines for e.g. decomposition. This type of refinement can also add new features to the machine too.

Data refinements are refinements where abstract states and rules are mapped to concrete ones in such a way that the effect of each concrete operation on concrete data types is the same as the effect of the corresponding abstract operation on abstract data types. Data refinements [27] are the basic refinements used in many formal methods, such as VDM, Z, and B. In ASM data refinement, ASM rules remain largely unchanged, except where they need to be adjusted to deal with the new data format, and only the abstract functions and predicates in the ASM rules are further specified.

3.6.3 Correctness Proofs

The ASM method is not geared towards a systematic system verification. Based on the general notion of ASM refinement, correctness verifications are usually carried out informally using the *commuting diagrams* [17], see Figure 3.1. More formal ways for correctness proofs have been well studied in a series of work by Schellhorn in [93, 92, 96].

In [92], a tool-based refinement correctness verification approach is presented. The general notion of refinement has been modified into two different notions of correct refinement, based on the comparisons of input/output behaviour and traces (runs), which are suitable for result computing and reactive systems, respectively.

The notion for *input/output behaviour comparison* focuses on finite runs only, and requires that, if ASM$'$ is said to refine ASM, for every finite trace $(st'_0, \ldots, st'_n)$ of the refined machine ASM$'$, and every related state st_0 of ASM with $INV(st_0, st'_0)$, there exists a finite trace $(st_0, \ldots, st_m)$ in ASM such that the output states st'_n, st_m are related.

The notion of *trace correctness* is similar to the general notion of refinement given in 3.6.1. It concerns both the finite and the infinite runs and the intermediate states of interest in the runs. It uses a coupling invariant $INV(x, x')$, representing the notion of equivalence between states, to relate the states of interest in the corresponding runs. The difference from the general notion is that it only requires corresponding runs found when an initial state st_0 can be found in ASM being related to the initial state st'_0. This difference (between general ASM refinement on one side, and input/output behaviour comparison and trace correctness on the other), makes it possible to capture incremental refinements, which are not ASM refinements in the general sense. Also it is clear that the trace-correctness refinement is stronger than the input/output correctness refinement.

Based on the two notions of correctness refinements, formulas expressed in DL (Dynamic Logic) have been developed for verification using the support tool KIV. A case study is presented to show how verifications of a Prolog interpreter is done using the formulas and KIV tool.

In [93], verification using generalised forward simulation is presented. In this work, four definitions of refinement correctness are given, which further breaks down the definitions in [92] into partial and total correctness. The partial and total correctness are similar to the extended refinement notion in [27], for ruling out implementation of terminating runs by nonterminating runs in the total correctness refinement. The idea of the verification approach is to allow two corresponding runs to be split into arbitrary commuting diagrams, such as $(m, n)-$, $(0, n)-$, $(m, 0)-$refinements using coupling invariants, then the verification of refinement correctness is reduced to the verification of obligation for the commutativity of a diagram. As the proof method propagates the invariant forward through traces, it takes the form of forward simulation. The generalised forward simulation is proven to be sound in the work.

Later in [96], Schellhorn has shown how a completeness proof for ASM refinement can be constructed, which is different from the one in data refinement [27] by combining forward and backward simulation. The later is not working for ASM refinement, since the notion of correctness refinement in ASM considers that termination of all ASM runs from a specific initial state should be preserved as an important property.

Using a simple example, we show how refinement correctness in ASM, particularly for infinite runs, can be established.

Example 3.2. Let us consider the following ASMs which compute the average of a (multi)set of integers, adopted from [29]:

```
𝔐 = values
      external newvalue, average
      main : values := values ∪ {newvalue};
             average := sum(values) / nr(values)
```

and the refined ASM

```
𝔐* = sum, nr
       external newvalue, average
       main : sum := sum + newvalue; nr := nr + 1;
              average := sum / nr
```

The sets of initial states for $\mathfrak{M}$ and $\mathfrak{M}^*$ are the singletons

$$IS = \{s_0 := \{values \mapsto \emptyset\}\}$$
$$IS^* = \{s_0^* := \{sum \mapsto 0, nr \mapsto 0\}\}$$

We now want to show that $\mathfrak{M}^*$ refines $\mathfrak{M}$ w.r.t. the abstraction predicate

$$\mathcal{A}(s, s^*) :\Leftrightarrow sum([values]^s) = [sum]^{s^*} \wedge nr([values]^s) = [nr]^{s^*}$$

which guarantees that the same averages are returned.

For this we can use forward simulation: the initial states s_0, s_0^* are corresponding, i.e., $\mathcal{A}(s_0, s_0^*)$ holds. For every corresponding pair of states s_i, s_i^* with $\mathcal{A}(s_i, s_i^*)$ and every successor state s_{i+1}^* of s_i^*, we can find a successor state s_{i+1} of s_i such that $\mathcal{A}(s_{i+1}, s_{i+1}^*)$ holds:

$$s_{i+1} := \{values \mapsto [values]^{s_i} \cup \{[sum]^{s_{i+1}^*} - [sum]^{s_i^*}\}\}$$

Thus, given any (finite or infinite) run of $\mathfrak{M}^*$, we can construct a corresponding run for $\mathfrak{M}$: let s_i^* denote the states in the run of ASM^*. Since for every "matching" triple of states s_i, s_i^*, s_{i+1}^* there exists at least one state s_{i+1} which completes the commutation diagram, there exists a function $f : \Sigma \times \Sigma^* \times \Sigma^* \to \Sigma$ which selects such a state s_{i+1} for every matching triple (in general this requires the axiom of choice (suggested by G. Schellhorn), but here we don't need it since s_{i+1} can be defined explicitly). This allows us to define a corresponding run of $\mathfrak{M}$ by induction: $s_{i+1} := f(s_i, s_i^*, s_{i+1}^*)$. By construction we have that $\mathcal{A}(s_i, s_i^*)$ and $s_i \xrightarrow{\mathfrak{M}} s_{i+1}$, which shows that we have indeed found a corresponding run.

□

3.6.4 Notions of Refinement: A Comparison with ASM refinement

For different reasons, general or specific, many notions of refinement have been defined in the literature - we shall mention just a few of them. In [123], a method using refinement for stepwise program development is presented. A definition of data refinement is given in [27], and then redefined using Z notation in [29]. A refinement notion for programs in pre-/post-condition style is given in [78]. ASM refinement notions are described in [17, 93], and the work in [88] addresses some limitations in the existing formalisms for process refinement.

For general reasons, we have the principle of substitutivity and the principle of improvement. The notion of refinement defined in [29] describes the ones using the principle of substitutivity as follows:

"it is acceptable to replace one program by another, *provided* it is impossible for a user of the programs to observe that the substitution has taken place. If a program can be acceptably substituted by another, then the second program is said to be a *refinement* of the first."

The definition of refinement in [78] represents the ones based on improvement. The view here is that a program works as a contract between client and programmer:

"We take the client's point of view in describing the negotiation: if program prog2 is better than program prog2 *for the client*, we write prog1 $\sqsubseteq$ prog2. That relation $\sqsubseteq$ between programs is called *refinement*: we say that prog2 refines prog1."

The improvements appear in the form of removing uncertainty and non-determinism which is clearly shown in [78], by weakening the precondition and strengthening the post-condition of the program. Here the program, as the contract, has a wider meaning which includes not just the final program code but the specifications too, so the difference can be seen by the users. Otherwise, if the improvements will not be seen by the user of the program, then the above two notions are the same.

The notion of data refinement in [27] defines the refinement relation by inclusion of the input/output mapping sets where the refined one is included in the original one, using a weakly commuting diagram. However, a side effect of this notion is that a program with empty input/output mapping is a refined program of any original program. This is caused by neglecting the nonterminating runs in the notion of refinement, a common concern in most of the notions of refinement, which is addressed in [27] by an extension called total correctness refinement.

The notion of ASM refinement is based on the operational semantics, i.e. runs, of the ASM rules, instead of the concrete syntax of the rules. It considers both finite and infinite runs and relates the corresponding runs by the intended equivalence notion between states of interest. It diverts from the principle of substitutivity, and particularly allows new features to be observable in the refined machines. This is because it uses the refinement, not just for facilitating replacement in the development, but as a way to document every intended design decisions and to prepare for future changes. To compare with data refinement in [27], the similarity is that their notions of refinement are both defined using the weakly commuting diagram, see Figure 3.2.

However, for data refinement, corresponding runs in the original and refined pro-

$$\begin{tikzcd}
\bullet \arrow[r, "AI"] \arrow[dr, "CI"'] & \bullet \arrow[r, "A_k"] & \bullet \arrow[r, dotted] & \bullet \arrow[r, "A_l"] & \bullet \arrow[r, "AF"] & \bullet \\
& \bullet \arrow[u, "\alpha"'] \arrow[r, "C_k"'] & \bullet \arrow[u, "\alpha"'] \arrow[r, dotted] & \bullet \arrow[u, "\alpha"'] \arrow[r, "C_l"'] & \bullet \arrow[u, "\alpha"'] \arrow[ur, "CF"'] &
\end{tikzcd}$$

data refinement

$$\begin{tikzcd}
\bullet \arrow[r] \arrow[d, dash, "IR"'] \arrow[dr, phantom, "\circlearrowright"] & \bullet \arrow[r] \arrow[dr, equal] & \bullet \arrow[r] \arrow[d, phantom, "\circlearrowright"] & \bullet \arrow[r] \arrow[dl, equal] \arrow[dr, phantom, "\circlearrowright"] & \bullet \arrow[r] \arrow[dr, equal] & \bullet \arrow[r] \arrow[dr, phantom, "\circlearrowright"] & \bullet \arrow[r] & \bullet \arrow[dl, equal] \arrow[d, phantom, "\circlearrowright"] \arrow[dr, dash, "OR"] & \\
\bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet \arrow[r] & \bullet
\end{tikzcd}$$

ASM refinement

Figure 3.2: data refinement vs. ASM refinement

gram/machine are always the same length, but of possibly different length in ASM refinement. Apart from this, the weakened refinement notion in ASM for result computation system, namely the partial input/output correctness refinement notion [93], and the notion of the data refinement in [27] essentially coincide. More detailed and complete comparison with data refinement is presented in [94].

Chapter 4

Typed Abstract State Machines

The ASM method provides an open conceptual framework which allows standard notions, techniques and notations for specific application systems to be integrated easily. In this thesis, the ASM method is applied in the design of data intensive systems, in particular, data warehouses and OLAP systems. In order to make the well-developed terminology in the database application area available to the designers, the notion of ASMs is extended with types, called Typed ASMs, TASM in short. With TASMs, the notion of schema refinement and the notion of strong schema refinement are introduced. This is used for capturing the notion of schema dominance, which is the base for a set of provably correctness rules in schema integration introduced in Chapter 6. As we have indicated in Chapter 2, our intention of typed ASM is quite modest compared to other typed ASMs introduced in the literature [28, 124]. We do not aim for a general purpose type system, but to provide the convenience of a specialized type system to the designers of data warehouses.

In the remainder of the chapter, first a type system is introduced, followed by the notion of states, transition rule and terms in TASM. Other features specific to databases, such as bulk update commands as special rules for efficiency reasons, and structural recursion constructs as special terms particularly for data aggregation are also proposed. We then introduce the notions of schema refinement and strong schema refinement. In the last part of the chapter, a constructive proof that TASMs are captured by ASMs is presented.

4.1 A Type System

A type system can be viewed as a collection of types, each of which represents a fixed set of values. Such type systems can be defined by base types with some constructors. The following is used as a *type system* to extend ASMs with types:

$$t = b \mid \{t\} \mid a : t \mid t_1 \times \cdots \times t_n \mid t_1 \oplus \cdots \oplus t_n \mid \mathbb{1}$$

Here b represents a not further specified collection of base types such as *label*, *id*, *ref*, *int*, *date*, etc. $\{\cdot\}$ is a set-type constructor, $a : t$ is a type constructor with a of type *label*, which is introduced as attributes used in join operations. The base type *id* is used for uniquely identifying tuples in a relation, *ref* is an alias for *id*, but is used for referencing tuples. Attribute names (i.e. the a in "$a : t$") are modelled explicitly as type *label* so we can easily refer to them (e.g. for storing FDs function dependencies). $\times$ and $\oplus$ are constructors for tuple and union types. $\mathbb{1}$ is a trivial type. With each type t we associate a *domain* $dom(t)$ in the usual way, i.e. we have

- $dom(\{t\}) = \{x \subseteq dom(t) \mid |x| < \infty\}$,
- $dom(a : t) = dom(t)$,
- $dom(t_1 \times \cdots \times t_n) = dom(t_1) \times \cdots \times dom(t_n)$,
- $dom(t_1 \oplus \cdots \oplus t_n) = \coprod_{i=1}^{n} dom(t_i) = \bigcup_{i=1}^{n} \{i\} \times dom(t_i)$,
- $dom(\mathbb{1}) = \{\mathbf{1}\}$

In addition, we assume an auxiliary base type U, whose domain includes all domains of types constructible without U. We will use this where it is more convenient not to specify the exact type yet (this can be done via later refinements). For further convenience, we will omit specifying U explicitly as type in declarations, and write just "a" instead of "$a : U$" for some label a, where this does not cause ambiguities.

For this type system we obtain the usual notion of subtyping, which is aligned with the subtyping notion in Object Orientation, that is, a super type has less information than its subtypes. Our subtype system is defined as the smallest partial order $\leq$ satisfying

- $t \leq \mathbb{1}$ for all types t;
- if $t \leq t'$ holds, then also $\{t\} \leq \{t'\}$;
- if $t \leq t'$ holds, then also $a : t \leq a : t'$;
- if $t_{i_j} \leq t'_{i_j}$ hold for $j = 1, \ldots, k$, then $t_1 \times \cdots \times t_n \leq t'_{i_1} \times \cdots \times t'_{i_k}$ for $1 \leq i_1 < \cdots < i_k \leq n$;
- if $t_i \leq t'_i$ hold for $i = 1, \ldots, n$, then $t_1 \oplus \cdots \oplus t_n \leq t'_1 \oplus \cdots \oplus t'_n$.

We say that t is a *subtype* of t' iff $t \leq t'$ holds. Obviously, subtyping $t \leq t'$ induces a canonical projection mapping $\pi^t_{t'} : dom(t) \to dom(t')$ when t' can be uniquely identified in t. This excludes ambiguous cases such as $t = int \times int$ and $t' = int$, where we do not know which int we should project onto. In such cases, we use labels (e.g. $t = a : int \times b : int$ and $t' = a : int$) to resolve this ambiguity.

Definition 4.1 (Minimal Common Supertype)**.** For two or more types $t_1, \ldots, t_n$ we call t a *minimal common supertype* of $t_1, \ldots, t_n$ if t is a common supertype of $t_1, \ldots, t_n$ and minimal w.r.t. the subtype ordering among all common supertypes.

While two (or more) types do not need to have a unique minimal common supertype, we can guarantee uniqueness by imposing a simple restriction on the way types are constructed.

Definition 4.2 (Labeled Component Property)**.** We say that a type t has the *labeled component* property, if all tuple or union constructs appearing in t are of the form $a_1 : s_1 \times \ldots \times a_n : s_n$ or $a_1 : s_1 \oplus \ldots \oplus a_n : s_n$, respectively, where the $a_i : s_i$ are labeled types with pairwise distinct labels a_i.

We will use tuple types to model relations, and label types to attach attribute names to the columns. Union types will be used to model clusters, so the types we use will always have the labeled component property.

Lemma 4.3. *Every pair of types t_1, t_2 has a minimal common supertype. If t_1, t_2 have the labeled component property, then this minimal common supertype is unique except for tuple orderings.*

Proof. The trivial type $\mathbb{1}$ is a common supertype of t_1, t_2, and it is easy to show that there can only exist a finite number of common supertypes. Thus there exists at least one minimal common supertype.

We proceed by induction on the number of type constructors used to construct t_1. The lemma is trivial for $t_1 = \mathbb{1}$ or base types t_1. We distinguish all other cases based on the outer type constructor used.

(1) Let $t_1 = \{t'_1\}$ for some type t'_1. If t_2 is not a set type, then the only common supertype of t_1, t_2 is $\mathbb{1}$. Otherwise $t_2 = \{t'_2\}$ for some type t'_2, and every common supertype t_S (other than $\mathbb{1}$) of t_1, t_2 is of the form $t_S = \{t'_S\}$, where t'_S is a common supertype of t'_1, t'_2. By our induction hypothesis there exists a unique (up to tuple orderings) minimal common supertype t'_{MS} of t'_1, t'_2, so $t_{MS} = \{t'_{MS}\}$ is the unique minimal common supertype of t_1, t_2.

(2) For $t_1 = a : t'_1$ argue analogous to (1).

(3) Let $t_1 = a_1 : s_1 \times \ldots \times a_n : s_n$ be a tuple type with pairwise different labels a_i. If t_2 is not a tuple type or labeled type, then the only common supertype is again $\mathbb{1}$. Otherwise we have
$$t_2 = a'_1 : s'_1 \times \ldots \times a'_m : s'_m$$
($m = 1$ if t_2 is a labeled type), and the a'_i are pairwise different. Thus we can re-index the tuple components of t_1, t_2 such that $a_i = a'_i$ for $i = 1 \ldots k$, and the remaining labels $a_{k+1}, \ldots, a_n, a'_{k+1}, \ldots, a'_m$ are pairwise different. Then every common supertype of t_1, t_2 is a supertype of
$$t_{MS} = a_1 : r_1 \times \ldots \times a_k : r_k$$
where r_i is the unique minimal common supertype of s_i, s'_i. Thus t_{MS} is the unique (up to tuple ordering) minimal common supertype of t_1, t_2.

(4) For $t_1 = a_1 : s_1 \oplus \ldots \oplus a_n : s_n$ argue analogous to (3).

□

Note that this result trivially extends to more than two types.

4.2 Signatures and States

The *signature* of a TASM is defined analogously to the signature of an "ordinary" ASM, i.e. by a finite list of function names $f_1, \ldots, f_m$. However, in a TASM each function f_i instead of an arity now has a *kind* $t_i \to t'_i$ involving two types t_i and t'_i. We interpret each such function by a total function $f_i : dom(t_i) \to dom(t'_i)$. Note that using $t'_i = t''_i \oplus \mathbb{1}$ we can cover also partial functions. Similar to ASMs, functions can be *dynamic* or *static*, and dynamic function can be controlled function or derived function.

The functions of a TASM including the dynamic and static functions, define the set of states of the TASM. More precisely, each pair $\ell = (f_i, x)$ with $x \in dom(t_i)$ defines a *location* with $v = f_i(x)$ as its *value*. Thus, each *state* of a TASM may be considered as a set of location/value pairs.

We call a function R of kind $t \to \{\mathbb{1}\}$ a *relation*. This generalises the standard notion of relation, in which case we would further require that t is a tuple type $a_1 : t_1 \times \cdots \times a_n : t_n$. In particular, as $\{\mathbb{1}\}$ can be considered as a truth value type, we may identify R with a subset of $dom(t)$, i.e. $R \simeq \{x \in dom(t) \mid R(x) \neq \emptyset\}$. In this spirit we also write $x \in R$ instead of $R(x) \neq \emptyset$, and $x \notin R$ instead of $R(x) = \emptyset$.

In addition to function declarations and rule definitions, a TASM also may contain *type definitions.* They take the form

$$\text{name} \ =_t \ \text{type}$$

where name is a type name (we assume a separate universe for type names, similar to function and rule names), and type a type expression. Type expressions are constructed from base types and already defined type names using the defined type constructors. Cyclic definitions are prohibited. Thus e.g. the definitions

$date =_t day : int \times month : int \times year : int$
$period =_t start : date \times end : date$

would be valid, but

$book =_t title : string \times authors : \{string\} \times references : \{book\}$

would not be, since book occurs in the type expression, creating a cycle. While it would certainly be possible to allow cyclic definitions, we chose to avoid them to keep our type system simple. This way, every legal type expression has an equivalent type expression containing only base types (i.e. no type names).

4.3 Transition Rules

As with ASMs we define state transitions via *update rules.* Most of these rules are defined the same as in ASMs, with the following exception - in the *Update, Let, Forall, Choose* and *Call* rules, we must have matching types:

- *Update Rule:*

  $f(\tau) := \tau'$

  Syntactic conditions:

  - f is a function of type $t \to t'$
  - τ is a term of type t
  - τ' is a term of type t'

- *Let Rule*

  `let` $x = t$ `in` R

  Syntactic conditions:

  - the type of x in R is the type of t

- *Forall Rule:*

  `forall` $x : type$ `with` φ `do` R

  Syntactic conditions:

  - the type of x in φ and R is *type*

- *Choose Rule:*

  `choose` x : *type* `with` φ `do` R

  Syntactic conditions:

  – the type of x in φ and R is *type*

- *Call Rule:*

  $r(\tau)$

  Syntactic conditions:

  – the type of τ matches the type in the declaration of r

Each update rule r defines an update set $\Delta(r)$ in the same way as for "ordinary" ASMs [17, p.74]. The notions of consistent update set, run, etc. are also exactly the same as for untyped ASMs.

Note that we may also omit the type declaration in the Forall and Choose rules. In this case, the type of x is understood to be U.

4.4 Terms

What is different in TASMs is that the terms used in the rules are typed, i.e. for each type t we obtain a set $\mathbb{T}_t$ of terms of type t. Then also the formulae φ used in the rules change in that equational atoms $\tau_1 = \tau_2$ can only be built from terms τ_1, τ_2 that have the same type. All the rest remains unchanged.

So let us assume that for each type t we are given a set V_t of variables of type t. Then we should have $V_t \subseteq \mathbb{T}_t$ and $dom(t) \subseteq \mathbb{T}_t$ (treating values as constant symbols which are interpreted as themselves), and further terms can be build as follows:

- For $\tau \in \mathbb{T}_t$ and $t \leq t'$ we get $\pi^t_{t'}(\tau) \in \mathbb{T}_{t'}$.
- For $\tau \in \mathbb{T}_{t_1 \times \cdots \times t_n}$ we get $\pi_i(\tau) \in \mathbb{T}_{t_i}$.
- For $\tau_i \in \mathbb{T}_{t_i}$ for $i = 1, \dots, n$ we get $(\tau_1, \dots, \tau_n) \in \mathbb{T}_{t_1 \times \cdots \times t_n}$, $\iota_i(\tau_i) = (i, \tau_i) \in \mathbb{T}_{t_1 \oplus \cdots \oplus t_n}$, and $\{\tau_i\} \in \mathbb{T}_{\{t_i\}}$.
- For $\tau_1, \tau_2 \in \mathbb{T}_{\{t\}}$ we get $\tau_1 \cup \tau_2 \in \mathbb{T}_{\{t\}}$, $\tau_1 \cap \tau_2 \in \mathbb{T}_{\{t\}}$, and $\tau_1 - \tau_2 \in \mathbb{T}_{\{t\}}$.
- For $\tau \in \mathbb{T}_{\{t\}}$, a constant $e \in dom(t')$ and static functions $f : t \to t'$ and $g : t' \times t' \to t'$ we get $src[e, f, g](\tau) \in \mathbb{T}_{t'}$, where $src[e, f, g]$ is a structured recursion, which is defined straight after the introduction of the terms.
- For $\tau_i \in \mathbb{T}_{\{t_i\}}$ for $i = 1, 2$ we get $\tau_1 \bowtie \tau_2 \in \mathbb{T}_{\{t_1 \bowtie t_2\}}$ using the maximal common subtype $t_1 \bowtie t_2$ of t_1 and t_2.
- For $x \in V_t$ and a formula φ we get $\mathbf{I}x.\varphi \in \mathbb{T}_t$.

The last three constructions for terms need some more explanation. Structural recursion $src[e, f, g](\tau)$ is a powerful construct, useful in particular for database queries, which is defined as follows:

- $src[e, f, g](\tau) = e$, if $\tau = \emptyset$;

- $src[e, f, g](\tau) = f(v)$, if $\tau = \{v\}$;
- $src[e, f, g](\tau) = g(src[e, f, g](v_1), src[e, f, g](v_2))$, if $\tau = v_1 \cup v_2$ and $v_1 \cap v_2 = \emptyset$.

In order to be uniquely defined, the function g must be associative and commutative with e as a neutral element.

We can use structural recursion to specify set comprehension, which is extremely important for views. We get $\{x \in \tau \mid \varphi\} = src[\emptyset, f_\tau, \cup](\tau)$ using the static function f_τ with $f_\tau(x) = \{x\}$, if $\varphi(x)$ holds, else $f_\tau(x) = \emptyset$, which can be composed out of very simple functions.

A simple example for structural recursion, other than set comprehension, would be summing up all numbers in a set. For any finite set $\mathcal{X}$ of numbers (e.g. integers), we have

$$\sum_{x \in \mathcal{X}} x = src[0, id, +](\mathcal{X})$$

where *id* is the identity function.

For the *join* $\tau_1 \bowtie \tau_2$, using $[\![\cdot]\!]_s$ to denote the interpretation in a state s, we get

$$\begin{aligned}[\![\tau_1 \bowtie \tau_2]\!]_s = \{v \in dom(t_1 \bowtie t_2) \mid \exists v_1 \in [\![\tau_1]\!]_s, v_2 \in [\![\tau_2]\!]_s.\\ (\pi_{t_1}^{t_1 \bowtie t_2}(v) = v_1 \wedge \pi_{t_2}^{t_1 \bowtie t_2}(v) = v_2)\},\end{aligned}$$

which generalises the natural join from relational algebra.

$\mathbf{I}x.\varphi$ stands for "the unique x satisfying φ". If such an x does not exist, the term will be undefined, i.e. we have $[\![\mathbf{I}x.\varphi]\!]_s = v$, if $[\![\{x \mid \varphi\}]\!]_s = \{v\}$, otherwise it is undefined.

Note: the Hilbert-generator $\mathbf{I}$ would be sufficient to cover all the other cases. Nevertheless, we keep the other constructs for ease of formulation, so our language is not minimised.

4.5 Bulk Updates

For relations R of kind $t \to \{\mathbb{1}\}$ we further permit *bulk assignments*, which take one of the following forms $R := \tau$ (for replacing), $R :+_k \tau$ (for inserting), $R :-_k \tau$ (for deleting), and $R :\&_k \tau$ (for updating), using each time a term τ of type $\{t\}$, and a supertype k of t (key of a relation). For deleting operation it suffices that τ is of type $\{t'\}$ for some "superkey" type t' with $k \leq t' \leq t$. These constructs are shortcuts for the following TASM rules:

- $R := \tau$ represents

```
   forall $x : t$ with $x \in \tau$ do $R(x) := \{\mathbf{1}\}$
|| forall $x : t$ with $x \in R \wedge x \notin \tau$ do $R(x) := \emptyset$
```

- $R :+_k \tau$ represents

```
forall $x : t$ with $x \in \tau \wedge \forall y(y \in R \Rightarrow \pi_k^t(x) \neq \pi_k^t(y))$ do $R(x) := \{\mathbf{1}\}$
```

- $R :-_k \tau$ represents

```
forall $x : t$ with $x \in R \wedge \exists y(y \in \tau \wedge \pi_k^t(x) = \pi_k^{t'}(y))$ do $R(x) := \emptyset$
```

- $R :\&_k \tau$ represents $R :-_k \tau; R :+_k \tau$

4.6 Schema Refinement in TASM

The intuition behind our notion of schema refinement is the following: our TASM consists of data and rules which operate on the data. Our refinements correspond to schema transformations, and therefore modify the way the data is represented, while rules stay essentially the same (although they need to be adapted where they access the data). Thus, instead of comparing runs in different TASMs directly, we restrict our attention to the way the schema is changed. We will show that if we have a schema refinement, then rules can be adapted in a straight-forward manner to obtain a refinement in the classical sense.

Recall that with each TASM $\mathfrak{M}$ we associate a schema $\mathcal{S}$, and denote the restriction of a state $s \in \Sigma$ onto $\mathcal{S}$ by $s[\mathcal{S}]$.

Definition 4.4. Let $\mathfrak{M}, \mathfrak{M}^*$ be two TASMs with schemas $\mathcal{S}, \mathcal{S}^*$. We say that $\mathfrak{M}^*$ is a *schema refinement* of $\mathfrak{M}$ w.r.t. an abstraction predicate $\mathcal{A}_S \subseteq \Sigma[S] \times \Sigma^*[S^*]$, if there exist computable functions $f : \Sigma[S] \to \Sigma^*[S^*]$ and $g : \Sigma^*[S^*] \to \Sigma[S]$ such that

$$f \subseteq \mathcal{A}_S \subseteq g^{-1} \tag{4.1}$$

Note that (4.1) implies that $g \circ f$ is the identity on $\Sigma[S]$. Also, the abstraction predicate $\mathcal{A}_S$ relates $\Sigma[S]$ and $\Sigma^*[S^*]$ rather than Σ and Σ^*. We say that two states $s \in \Sigma, s^* \in \Sigma^*$ are corresponding, iff $s[S], s[S^*]$ are related via $\mathcal{A}_S$. This convention allows us to also talk about refinements (rather than schema refinements) w.r.t. the induced abstraction predicate $\mathcal{A}$ on $\Sigma \times \Sigma^*$:

$$\mathcal{A} := \{(s, s*) \in \Sigma \times \Sigma^* \mid (s[S], s^*[S^*]) \in \mathcal{A}_S\}$$

We will show next how any schema refinement can be extended to a total refinement, by adapting the rules to work with the new schema instead. Recall that $\mathfrak{M}^*_S$ is a total refinement of $\mathfrak{M}$ if $\mathfrak{M}^*_S$ and $\mathfrak{M}$ refine each other.

Theorem 4.5. *Let $\mathfrak{M}^*_S$ be a schema refinement of $\mathfrak{M}$ w.r.t. $\mathcal{A}_S$. Then $\mathfrak{M}^*_S$ can be modified in a manner which leaves S^* unchanged, to obtain a total refinement $\mathfrak{M}^*$ of $\mathfrak{M}$ w.r.t. $\mathcal{A}$.*

Proof. Let f, g be as in Definition 4.4. The new TASM $\mathfrak{M}^*$ is constructed as follows. We start with a copy of $\mathfrak{M}$, add the relations in $S^* \setminus S$ (if $\mathfrak{M}$ already contains such relations, rename them), and define the schema of $\mathfrak{M}^*$ as S^*. We keep the relations in $S \setminus S^*$, but do not make them part of our schema. They are preserved to temporarily store the data in its original form.

Now let $\hat{f}, \hat{g}$ be the functions obtained when "extending" f, g to functions on Σ, Σ^* by leaving all TASM functions *not* in $S \cup S^*$ unchanged.

We then add two new rules **revert** and **convert**, which implement the functions $\hat{g}$ and $\hat{f}$, respectively. Since both functions are computable (since f, g are), this is always possible. The **main** rule is then adapted to

$$\mathbf{main}_{new} := \mathbf{revert};\ \mathbf{main}_{old};\ \mathbf{convert}$$

i.e. we first revert the data back to its original format, then run the old main rule in its original form, and convert the data back to its new format afterwards.

Finally, we define the set of initial states of $\mathfrak{M}^*$ as $IS^* := \hat{f}(IS)$, where IS denotes the initial states of $\mathfrak{M}$. It is now easy to show that $\mathfrak{M}^*$ refines $\mathfrak{M}$, with abstraction predicate $\mathcal{A}$, using forward simulation. For this we strengthen our abstraction predicate to $\hat{\mathcal{A}}$ as follows:

$$\hat{\mathcal{A}} := \{(s, s^*) \in \mathcal{A} \mid s[\overline{S \cup S^*}] = s^*[\overline{S \cup S^*}]\}$$

where $\overline{S}$ denotes the complement of S. Note that from (4.1) it follows that

$$\hat{f} \subseteq \hat{\mathcal{A}} \subseteq \hat{g}^{-1}$$

This allows us to show the stronger statement that $\mathfrak{M}^*$ refines $\mathfrak{M}$ w.r.t. $\hat{\mathcal{A}}$ instead.

Now let i, i^* be two corresponding states of $\mathfrak{M}, \mathfrak{M}^*$, i.e., we have $(i, i^*) \in \hat{\mathcal{A}}$, and let o^* be obtained by running $\mathfrak{M}^*$ in state i^*. By definition of composition, we can "split" the application of $\mathfrak{M}^*$ into 3 steps with intermediate states i', o':

$$i^* \xrightarrow{\textbf{revert}} i' \xrightarrow{\textbf{main}_{old}} o' \xrightarrow{\textbf{convert}} o^*$$

Since $(i, i^*) \in \hat{\mathcal{A}} \subseteq \hat{g}^{-1}$, and by construction **revert** implements $\hat{g}$, the states i, i' are identical on all TASM functions in $\mathfrak{M}$. We therefore can find a state $o \in \Sigma$ which is identical to o' on all functions in $\mathfrak{M}$, and obtained from i by executing $\mathfrak{M}$. Since **convert** implements $\hat{f}$ we have $(o, o^*) \in \hat{f} \subseteq \hat{\mathcal{A}}$, which shows that forward-simulation w.r.t. $\hat{\mathcal{A}}$ is possible.

The proof that $\mathfrak{M}$ refines $\mathfrak{M}^*$ is done in similar manner. □

While the refinement described in the proof above is not the most efficient one, optimizations which modify the TASM rules to operate on the data in its new form directly can be introduced in further refinement steps. Note that these further refinements are *not* covered by our rules. Doing so is not feasible, since access to our data cannot easily be restricted to a finite set of operations, as is often done for simpler data structures such as e.g. stacks or lists [..]. It might be possible to do so for a single relation which is decomposed into multiple "subrelations" (e.g. allowing operations add_tuple, remove_tuple, check_contains_tuple), but in general our schema contains many relations, and computations performed on the schema are too complex to capture with a pre-defined set of operations.

We note further that, while the functions f, g in Definition 4.4 have to be computable, they and the abstraction predicate $\mathcal{A}_S$ can still be quite arbitrary. In principal it is possible to refine an entire schema into a single integer variable via "clever" computable transformations, although this is hardly the type of refinement we have in mind. We therefore wish to restrict $\mathcal{A}_S$ further to the type of predicates which occur typically in schema transformations.

We first define isomorphisms starting from bijections $\alpha_b : dom(b) \to dom(b)$ for all base types b. This can be extended to bijections α_t for any type t as follows:

$$\begin{aligned}
&\alpha_{t_1 \times \cdots \times t_n}(x_1, \ldots, x_n) = (\alpha_{t_1}(x_1), \ldots, \alpha_{t_n}(x_n)) \\
&\alpha_{t_1 \oplus \cdots \oplus t_n}(i, x_i) = (i, \alpha_{t_i}(x_i)) \\
&\alpha_{\{t\}}(\{x_1, \ldots, x_k\}) = \{\alpha_t(x_1), \ldots, \alpha_t(x_k)\}
\end{aligned}$$

Then α is an *isomorphism* of $\mathcal{S}$ iff for all states s, the permuted state $\alpha(s)$, and all $R : t \to \{\mathbb{1}\}$ in $\mathcal{S}$ we have $R(x) \neq \emptyset$ in s iff $R(\alpha_t(x)) \neq \emptyset$ in $\alpha(s)$.

We can now strengthen our notion of schema refinement as follows.

Definition 4.6. A schema refinement with abstraction predicate $\mathcal{A}_S$ is a *strong schema refinement* if $\mathcal{A}_S$ is *invariant under isomorphisms*, that is

$$(s, s^*) \in \mathcal{A}_S \iff (\alpha_S(s), \alpha_{S^*}(s^*)) \in \mathcal{A}_S$$

for all isomorphisms α_S, α_{S^*} induced by the same bijections on base types.

The intuition behind this definition is the following: functions invariant under isomorphisms can only compare and copy basic attribute values, and thus only perform operations which could be considered "reasonable" database transformations. On the other hand, most database transformation functions (e.g. all relational algebra operators except constant selection) are invariant under isomorphisms. Correspondingly, almost all of our refinement rules produce a strong schema refinement.

For all of our refinement rules, we will provide the abstraction predicate $\mathcal{A}_S$ (usually just denoted as $\mathcal{A}$, as this won't cause any ambiguities), as well as the functions f and g. In principal, to show that the transformations are indeed (strong) schema refinements, we would have to prove that f, g are computable (and possibly that $\mathcal{A}_S$ is invariant under isomorphisms), and that $f \subseteq \mathcal{A}_S \subseteq g^{-1}$ holds. However, all these conditions will be obvious for the refinement rules we introduce, so that providing $\mathcal{A}_S, f, g$ is sufficient.

Most of of our rules always create strong schema refinements. The only possible exceptions are Rules 15 and 21, which provide strong schema refinements iff the predicate φ is invariant under isomorphisms, as well as Rules 18-20, which produce strong schema refinements iff the function h is invariant under isomorphisms.

4.7 An Equivalence Result

Let us now look at the relationship between TASMs and ASMs. We will show how to translate a TASM $\mathfrak{M}$ into an ASM $\Phi(\mathfrak{M})$.

Functions are simply translated by removing their type, and type definitions are removed altogether. If desired, a function $f : t \to t'$ in $\mathfrak{M}$ with tuple type $t = t_1 \times \cdots \times t_n$ as parameter can be translated into a function $\hat{f}$ in $\Phi(\mathfrak{M})$ with n separate parameters, but this is optional.

We then have to translate the rules in $\mathfrak{M}$. Here we get a one-one correspondence between rule constructs in ASM and TASM, except for bulk update operators, which however are just shortcuts and can be replaced using their definition. Thus we only need to worry about two things: type checking and translation of terms.

Type checking is only necessary for the `forall` and `choose` rules, since all other type restrictions are static. For dynamic type checking we define some auxiliary unitary functions tc_t which return true iff their argument is of type t. Here we must assume that $tc_b(x)$ can be checked for all base types b. Based on this, it is straight forward to construct $tc_t(x)$ for complex types:

$$\begin{aligned}
tc_{\{t\}}(x) :=& \quad \exists k.x = \{x_1, \dots, x_k\} \wedge \forall x_i \in x.tc_t(x_i) \\
tc_{a:t}(x) :=& \quad tc_t(x) \\
tc_{t_1 \times \cdots \times t_n}(x) :=& \quad x = (x_1, \dots, x_n) \wedge \bigwedge_{1 \le i \le n} tc_{t_i}(x_i) \\
tc_{t_1 \oplus \cdots \oplus t_n}(x) :=& \quad \bigvee_{1 \le i \le n} (x = (i, x_i) \wedge tc_{t_i}(x_i)) \\
tc_{\mathbb{1}}(x) :=& \quad x = \mathbf{1}
\end{aligned}$$

We can now translate the rule constructs

`forall` $x : \textit{type}$ `with` φ `do` R
`choose` $x : \textit{type}$ `with` φ `do` R

in TASM into the following ASM rules, where $\hat{\varphi}, \hat{R}$ are the translations for φ, R:

```
forall $x$ with $tc_{type}(x) \wedge \hat{\varphi}$ do $\hat{R}$
choose $x$ with $tc_{type}(x) \wedge \hat{\varphi}$ do $\hat{R}$
```

The problem that remains is to translate the complex term language of TASMs into the ASM term language.

For subtype projection functions $\pi^t_{t'}$ we may assume these are defined in the signature of $\Phi(\mathfrak{M})$ by auxiliary functions. Giving an inductive definition for these (similar to the type checking functions) is not hard but lengthy, so we will skip it here. The projection function π_i is just a special case of subtype projection function. A term σ containing $\iota_i(\tau)$ can be replaced by $\sigma[\iota_i(\tau)/(i,\tau)]$.

If a rule r in $\mathfrak{M}$ involves a term $\mathbf{I}x.\varphi$, we replace it using the choice rule:

```
choose $x$ with $\varphi$ do $r[\mathbf{I}x.\varphi/x]$
```

Here we must take care with the sequential operator ; to ensure that φ is interpreted in the correct state. If $r = r_1; r_2$ we perform the above replacement for r_1 and r_2 separately, since r_1 might change the value at locations occurring in φ.

Now consider structural recursion, which presents the most complicated case. For every partial term $src[e,f,g]$ occurring in a rule of $\mathfrak{M}$, we add a controlled function $SRC_{e,f,g}$ to $\Phi(\mathfrak{M})$, which we will use to store the mapping induced by $src[e,f,g]$. Furthermore, we add a unary rule $computeSRC_{e,f,g}$ to $\Phi(\mathfrak{M})$, which populates $SRC_{e,f,g}$ as follows:

```
$computeSRC_{e,f,g}(x) =$
  if $SRC_{e,f,g}(x) = undef$ then
    if $x = \emptyset$ then $SRC_{e,f,g}(x) := e$
    else if $\exists z.x = \{z\}$ then
      choose $z$ with $x = \{z\}$ do $SRC_{e,f,g}(x) := f(z)$
    else choose $z1, z2$ with $x = z_1 \cup z_2 \wedge z_1 \cap z_2 = \emptyset \wedge z_1 \neq \emptyset \wedge z_2 \neq \emptyset$ do
      $computeSRC_{e,f,g}(z_1); computeSRC_{e,f,g}(z_2);$
      $SRC_{e,f,g}(x) := g(SRC_{e,f,g}(z_1), SRC_{e,f,g}(z_2))$
```

Whenever a rule r in $\mathfrak{M}$ involves the term $src[e,f,g](\tau)$, we now replace it by

$$computeSRC_{e,f,g}(\tau);\;\; r[src[e,f,g](\tau)/SRC_{e,f,g}(\tau)]$$

Again, if $r = r_1; r_2$ we perform the above replacement for r_1 and r_2 separately, to ensure that τ is interpreted to the same value in both parts of the above rule.

Finally, the join is just a special case of set comprehension (see its definition), which in turn can be expressed using structural recursion, and thus is covered as well.

Taking together these translations of rules, a rule r and its translation $\Phi(r)$ result in the same update set. Together with the canonical correspondence between states of $\mathfrak{M}$ and $\Phi(\mathfrak{M})$ we have obtained the following theorem.

Theorem 4.7. *For each TASM $\mathfrak{M}$ there is an equivalent ASM $\Phi(\mathfrak{M})$.*

Of course, this theorem also follows immediately from the main results on the expressiveness of ASMs in [10, 44]. However, what we need in our specific application area is also constructiveness, i.e. in order to fully exploit the theory of ASMs for data warehousing and OLAP systems we need a constructive translation as the one indicated above.

Chapter 5

Data Warehouse Design Using the ASM Method

The ASM method presumes to start with the definition of a *ground model ASM* (or several linked ASMs), which captures requirements, similar to scenarios in [98] or use-cases in [55, 90]. All further system development is done by refining the ASMs using the general refinement method.

The ground model captures the user requirements at a level that is close to the domain of the to-be-built system, based on clear and elegant operational semantics, which enables model checking by inspection or validation. The validation can be supported by execution tools or using the program walkthrough technique. The possibility for model inspection or validation is one of the desired properties for system design as it is impossible to mathematically verify the system against informal user requirements. Furthermore, the ground model allows us to do formal verification when system properties are formalised. Some examples for formal verifications can be found in [45], and many more on the ASM website [49].

The notion of refinement in the ASM method is not restrained by substitutivity, but supports introduction of new features into the system. It is a method for not only moving the model from abstract to concrete but also recording the intended design decisions in a precise way, which facilitates divide-and-conquer for overcoming the complexity of the design process.

Based on the ASM method, we provide a tailored refinement-based design framework for data warehouse and OLAP system design, which breaks down the design by concerns that are specific to the application, such as incremental design, view materialisation for query performance, data distribution design, etc. We apply the view/schema integration technique to overcome the integrity issue in our incremental design, which starts from one business process, namely a single datamart, and expands to more datamarts using pure incremental refinements. This is further supported by a set of correct rules for schema integration introduced in Chapter 6.

In Section 5.1, we first present a general ground model for a data warehouse and OLAP system, which is then applied in a grocery store example. This is first presented in ASM and then in TASM, to illustrate the advantages of TASMs. We will also show how some formalised properties of the ground model can be verified. Section 5.2 is used to describe a refinement process that is tailored for data warehouse and OLAP system design. Based on the refinement process, example refinements are shown in Section 5.3 with correctness proof based on the notion of ASM refinement.

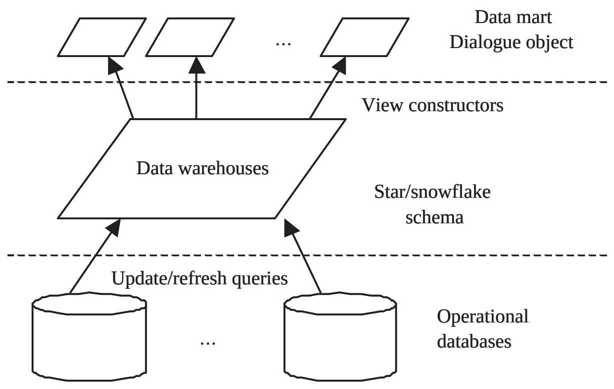

Figure 5.1: The general architecture of a data warehouse and OLAP

5.1 The Data Warehouse Ground Model

The basic idea of data warehousing is to separate the source data, which comes from operational databases, from the target data, which is used by OLAP systems. While doing so, we must keep the data warehouse in sync with the source data by regular updates ([117]). From this viewpoint, we may specify the basic requirements of a data warehouse and OLAP system as follows:

- the entire system consists of three components, namely the data source, also called the *operational database*, the *data warehouse* and the *OLAP* system, which form the three-tier architecture shown in Figure 5.1;
- in the operational database tier, the source data is modelled, and updates to the data warehouse is performed upon system request;
- in the data warehouse tier, the data warehouse data and the data extraction are modelled, and OLAP queries are answered upon system request;
- in the OLAP tier, the OLAP users and queries are modelled and managed.

5.1.1 The Operational Database ASM

We model the operational database tier by DB ASM. The basic requirements for DB ASM is to model the source data and the updates to the data warehouse upon request. Assuming for simplicity that all data sources are relational, the ASMs signatures would describe the relation schemata. For example, an n-nary relation R will be modelled as a boolean function $R(n)$. Hence, we will define a set of boolean functions for modelling the database relations. Further, we introduce a universe **request** to model the system requests on data warehouse updates or OLAP queries. We use an external function **req**:*request* for representing the current request to be served. Furthermore, we define a static external function **r-type**:*request*$\rightarrow$ $\{$ *extract, open-datamart*, $\dots\}$ to map the current request to its type. Here we use *extract* and *open-datamart* for they are common terms in data warehouse. Furthermore, the functions S_i which model the data warehouse data relations and the extraction rules *extract_*S_i are imported from the data warehouse model for updating the data warehouse. In addition, the functions for the operational database relations R_i are exported for constructing the extraction rules in data warehouse.

A general model for the data source DB-ASM is defined as follows:

```
ASM DB-ASM
IMPORT DW-ASM($S_1$, ..., $S_m$, $extract\_S_1$, ..., $extract\_S_m$)
EXPORT $R_1$, ..., $R_l$
SIGNATURE
  $R_1(ar_1)$, ..., $R_l(ar_l)$,
  r-type(1) (static), req(0) (external)
BODY
  main = if r-type(req)= extract then
    $extract\_S_1 || \ldots || extract\_S_m$
  endif
```

The DB ASM is straightforward. It can be shown that it captures the basic requirements by analysing the model line by line as follows:

- in the "IMPORT", we have imported the functions for data warehouse relations, S_1, ..., S_m, and the extraction rules *extract_*S_1, ..., *extract_*S_m, which are the queries defined in the data warehouse for extracting data from the data source. These are used to update the data warehouse upon system request.
- in the"EXPORT", we have the source data $R_1(ar_1)$, ..., $R_l(ar_l)$ exported so that they can be used in the data warehouse for defining the queries for data extraction.
- in the "SIGNATURE", we model the source data relations as $R_1(ar_1)$, ..., $R_l(ar_l)$. In addition, we model the system request by two external functions. The reason why external functions are used is that we do not want to be tied down by the design details of how data warehouse should be updated, such by system clock, or by changes in the data source, at this initial stage. The external functions will be dealt with in later refinements.
- in the main rule, the current system request is checked, and if it is of type "extract", the data warehouse is updated by executing the extraction rules at one atomic step.

5.1.2 The Data Warehouse ASM

We model the data warehouse tier by DW ASM. The basic requirements for the data warehouse module is to model the data warehouse data and the data extraction rules using the operational database, and to answer OLAP queries.

For the data warehouse ASMs we follow the same line of abstraction as for the operational databases, i.e. using boolean functions to model the data warehouse relations. In particular, we have the following:

- We define boolean functions S_1, ..., S_m for the data warehouse relations. For each data warehouse relation, we define a data extraction rule, extract_S_i, over the operational database relations. We export these functions and extraction rules for constructing view creation rules in OLAP ASM.
- We use the same external functions **req** and **r-type** as defined in DB ASM to handle the external requests.

- For answering OLAP queries, we define a rule *open-datamart*, and import functions V_i and rules *create_*V_1 from OLAP ASM.
- To cater for the fact that the same OLAP query may be requested from multiple users or from the same user multiple times, we use function DM_V_i imported from OLAP ASM to hold data requested by a specific user request using a unique identifier *dm*. In addition, functions, such as the-matching-view for getting the specific OLAP view and the-datamart for getting the identifier *dm* are imported from OLAP ASM for OLAP query process.

The DW-ASM is defined as follows:

```
ASM DW-ASM
IMPORT
  DB-ASM($R_1$, ..., $R_l$)
  OLAP-ASM($V_1$, ..., $V_n$, DM_$V_1$, ..., DM_$V_n$,
  create_$V_1$, ..., create_$V_n$, the-matching-view, the-datamart)
EXPORT $S_1$ ,..., $S_m$,
  extract_$S_1$, ..., extract_$S_m$
SIGNATURE
  $S_1(ar_1)$, ..., $S_m(ar_m)$,
  r-type(1) (static), req(0) (external)
BODY
  main = if r-type(req)=open-datamart then
    open_datamart(the-datamart(req)) endif

  extract_$S_1$ = ...
  ...
  extract_$S_m$ = ...

  open_datamart(dm) = case the-matching-view(dm) of
   $V_1$ : create_$V_1$;
      forall $a_1, \ldots, a_{ar_1}$ with
      $V_1(a_1, \ldots, a_{ar_1}) = 1$ do
         DM-$V_1(dm, a_1, \ldots, a_{ar_1}) := 1$ enddo
   ...
   $V_n$ : create_$V_n$;
      forall $a_1, \ldots, a_{ar_n}$ with
      $V_n(a_1, \ldots, a_{ar_n}) = 1$ do
         DM-$V_n(dm, a_1, \ldots, a_{ar_n}) := 1$ enddo
   endcase
```

The data warehouse model DW ASM is slightly different from DB ASM, as it interacts with both the data source and the OLAP system. Nevertheless, we can still show that it captures the requirements by checking the model line by line:

- in the "IMPORT", relations R_1, ..., R_l are imported from the data source, for constructing data extraction in data warehouse. The views V_1, ..., V_n, the datamarts DM_V_1, ..., DM_V_n, and the queries *create_*V_1, ..., *create_*V_n, which extract data from data warehouse for the views, are imported from the OLAP ASM for handling OLAP queries. In addition, two functions the-datamart, which maps the open-datamart request to a specific datamart, and the-matching-view, which maps the datamart to a specific view, are exported for use in answering individual OLAP queries.

- in the "EXPORT", data warehouse relations and data extraction queries are exported for updating the data warehouse in the DB ASM and constructing data extraction in the OLAP.
- in the "SIGNATURE", data warehouse data is modelled by $\mathrm{S}_1(ar_1)$, …, $\mathrm{S}_m(ar_m)$, white the current system request is modelled by two external functions r-type (external), req (external), which are the same functions used in DB ASM.
- in the main rule, the current request is checked. If it is of type "open-datamart", then rule *open_datamart* is called with the parameter "dm" passed by the-datamart(req).
- in the rule *open_datamart*(dm), the corresponding view to the datamart "dm" is created by calling the respective rule *create_*V_i, and then copied to the datamart DM-V_i for the specific user query request identified by "dm".
- the queries for data warehouse data extraction are defined by the rules *extract_*S_1, …, *extract_*S_m.

5.1.3 The OLAP ASM

We model the OLAP tier by OLAP ASM. The basic requirements for the OLAP module is to model and manage the users and the user queries.

We apply the concept of dialogue objects for managing the user queries. More details about dialogue objects are given in Section 6.6. The general idea of dialogue objects from [99] is that each user has a collection of open dialogue objects, i.e. opened datamarts in our case. At any time we may get new users by the operation "login", or the users may create new dialogue objects by "open", without closing the opened ones, or they may close some of the dialogue objects, or quit when they finish with the system. Figure 5.2 illustrates the process of management of users and user queries in OLAP-ASM. Other functionalities of OLAP include operations, such as roll-up, drill-down, slice and dice, over the opened datamarts. For a simple ground model, we leave these functions to refinements under requirements capturing, which is discussed in Section 5.2.

In the ground model of OLAP ASM, we define the universe **user** to model the user of the system, the universe **datamart** to model the opened datamarts, the universe **view** to model the views for the OLAP queries, and the universe **operation** to model the OLAP operations issued. Over the universes, we define the function **o-type**:**operation** $\rightarrow \{login, open, close, quit\}$, the function **owner**: **datamart** $\rightarrow$ **user**, the function **issuer**: **operation** $\rightarrow$ **user**, the function **the-datamart**:**operation** $\rightarrow$ **datamart**, which returns the specific datamart over which an operation is performed, the function **the-view**:**operation** $\rightarrow$ **view**, and the function **the-matching-view**:**datamart** $\rightarrow$ **view** which gives the matching view of the datamart. We use an external function **op**:operation to represent the current operation processed, **registered**:**user** for the logged-on users, and a set of functions $\mathbf{V}_i$ $(i = 1, \ldots, n)$ to define the views for the OLAP queries defined in the system. With each of the views we define a function **DM-**$\mathbf{V}_i$ to model the datamarts corresponding to the views $\mathbf{V}_i$, and holds the opened views for requests from multiple

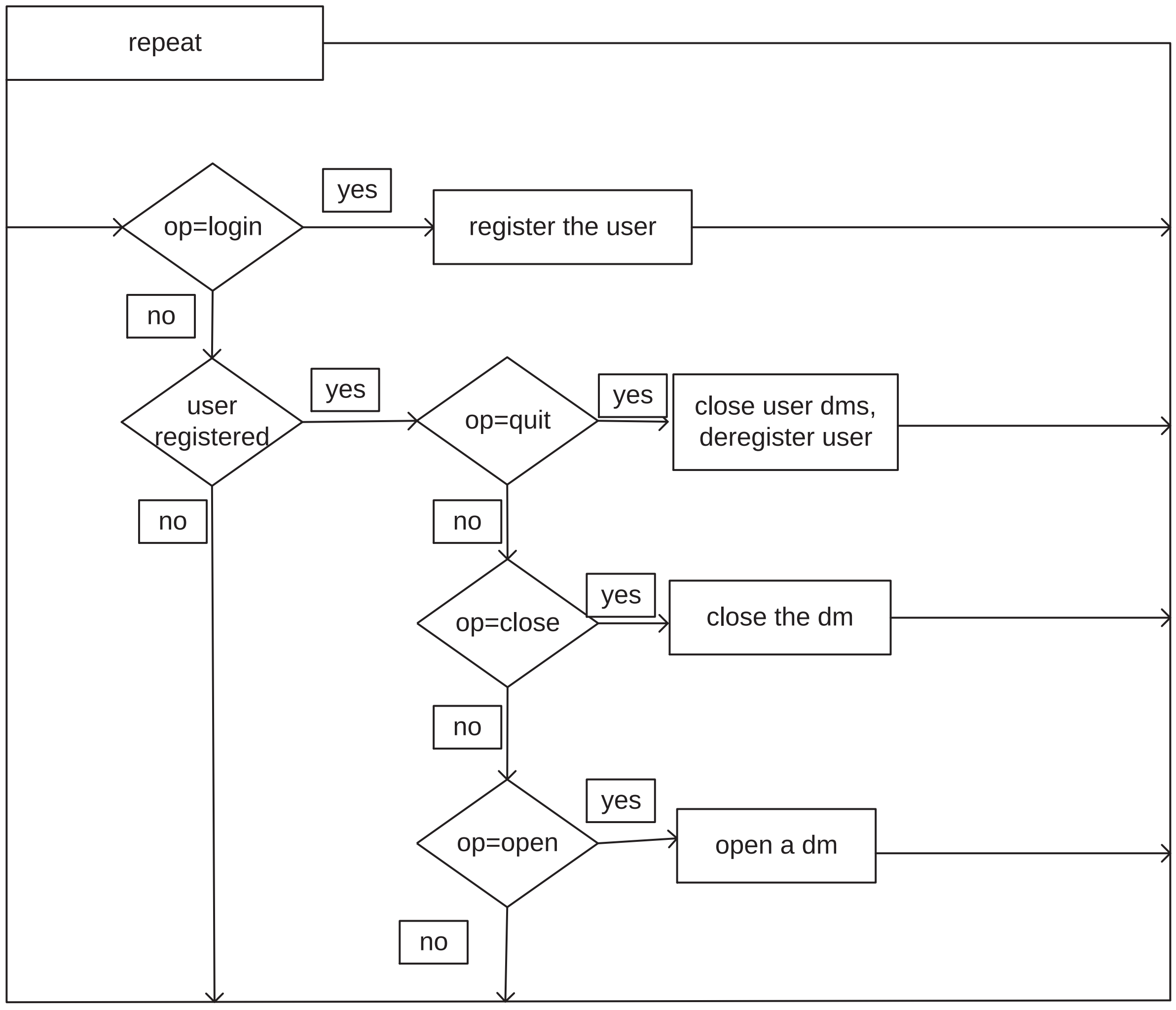

Figure 5.2: The main process in OLAP-ASM

users and multiple requests from the same users. We use the universe **request** to add the requests for opening datamarts in the system. Furthermore, the data warehouse relations are imported, and the OLAP views, the related functions, and the extraction rules are exported.

In ASMs, the OLAP-ASM is defined as follows:

```
ASM OLAP-ASM
IMPORT DW-ASM($S_1$, ..., $S_m$)
EXPORT $V_1$, ..., $V_n$, DM-$V_1$, ..., DM-$V_n$, the-matching-view, the-datamart,
    $create\_V_1$, ..., $create\_V_n$
SIGNATURE
  $V_1(ar_1)$, ..., $V_n(ar_n)$,
  DM-$V_1(ar_1 + 1)$, ..., DM-$V_n(ar_n + 1)$,
  o-type(1) (static),
  owner(1),
  issuer(1),
  registered(1),
  the-datamart(1),
  the-view(1),
  the-matching-view(1),
  op(0) (external)
BODY
  $main$ = if o-type($op$) = $login$ then $LOGIN$
      elsif if registered(issuer($op$))=1 then
        if o-type($op$) = $open$ then $OPEN$
      elsif o-type($op$) = $close$ then $CLOSE$
      elsif o-type($op$) = $quit$ then $QUIT$
  endif

  $LOGIN$ =
      registered(issuer($op$)):=1

  $OPEN$ =
      import $dm$
        datamart($dm$) := 1
        owner($dm$) := issuer($op$)
        the-matching-view($dm$) := the-view($op$)
        import $reqst$
          let $reqst$ = ($open\text{-}datamart$,dm) in
          request($reqst$) := 1
        end-import;
      end-import;

  $CLOSE$ =
      $close\_datamart$(the-datamart($op$));
      owner(the-datamart($op$)) := $\perp$
      datamart(the-datamart($op$)) := $\perp$

  $QUIT$ =
      let $usr$ = $issuer(op)$ in
        forall $dm$ with owner($dm$) = $usr$
          do $close\_datamart(dm)$
            datamart($dm$) := $\perp$ owner($dm$) := $\perp$ enddo
        registered($usr$) := $\perp$

  $close\_datamart(dm)$ =
    case $the - matching - view(dm)$ of
    $V_1$ : forall $a_1, \dots, a_{ar_1}$ with
      DM-$V_1(dm, a_1, \dots, a_{ar_1}) = 1$
      do DM-$V_1(dm, a_1, \dots, a_{ar_1})$ := $\perp$ enddo
    ...
    $V_n$ : forall $a_1, \dots, a_{ar_n}$ with
      DM-$V_n(dm, a_1, \dots, a_{ar_n}) = 1$
      do DM-$V_n(dm, a_1, \dots, a_{ar_n})$ := $\perp$ enddo
    endcase

  $create\_V_1 = \ldots$
    ...
  $create\_V_n = \ldots$
```

The OLAP ASM is slightly more complicated than the other two ASMs. The similarity with the other two ASMs is that it models the data and data extraction in the same way. The difference from the other two ASMs is that it also deals with user operations. Instead of checking the model line by line as above, we will focus on the rules, and show that it captures the requirements by analysing the rules against Figure 5.2, the description of the main process of the model.

- The main rule models the main process of the model.
- In the main rule we process the operations "LOGIN" and "QUIT", which are used for managing the users, as well as "CLOSE" and "OPEN", which are used for managing the user queries. This matches the set of operations defined in Figure 5.2.
- If current operation *op* is of type "LOGIN", rule *LOGIN* is called, which registers the user. This matches "register the user" under condition "op=login" in Figure 5.2.
- If the user is registered and the current operation *op* is of type "QUIT", rule *QUIT* is called, which closes all the datamarts that belong to the user, removes all the datamarts of the user from the list of opened datamarts, removes corresponding ownerships and deregister the user from the system. This matches "close user dms and deregister user" under conditions "user registered" and "op=quit" in Figure 5.2.
- If the user is registered and the current operation *op* is of type "CLOSE", rule *CLOSE* is called, which closes the datamart by setting it to undefined, removes the ownership of the datamart *dm* from the user, and removes the datamart *dm* from the list of opened datamarts. This matches "close the dm" under conditions "user registered" and "op=close" in Figure 5.2.
- If the user is registered and the current operation *op* is of type "OPEN", rule *OPEN* is called, which generates a unique id *dm* for a datamart, sets the owner of the datamart to be the user, the matching view of the datamart to be the view in the "OPEN" operation, and issues an "open-datamart" request. This matches "open a dm" under conditions "user registered" and "op=open" in Figure 5.2.

5.1.4 The Grocery Store Data Warehouse - a Simple Example

Consider a grocery store chain data warehouse, an example adopted from [71, p.358]. Assume the first datamart is for the sales process. For simplicity, the OLAP query is formulated as the total sales by shop, month and year. Its underlying schema is shown in the HERM (Higher-order Entity-Relationship Model, [106]) diagram in Figure 5.3. The data schema for the data source is shown in the HERM diagram in Figure 5.4. There are five relations, namely, Shop_DB, Product_DB, Customer_DB, Buys and Offer, of arity 4, 3, 4, 5 and 5, respectively. Based on the OLAP query, the schema for the data warehouse, a star schema [58], is shown in Figure 5.5. It is represented by a relational database

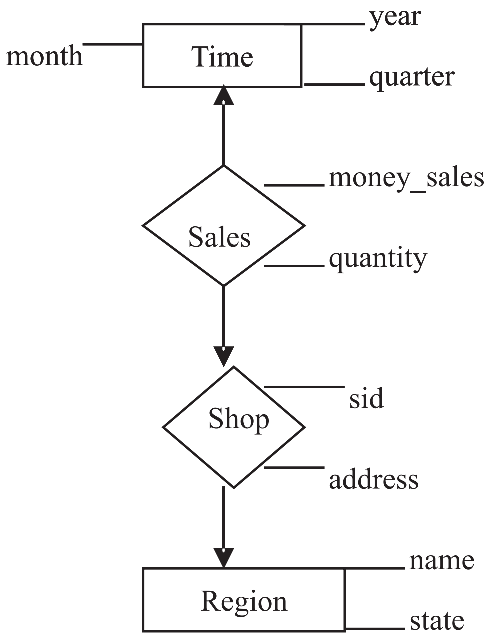

Figure 5.3: Schema underlying an OLAP view

schema with four relation schemata Shop, Product, Purchase and Time with arities 6, 3, 7 and 7, respectively. As shown in Figure 5.5, cid is populated in Purchase for keeping the data at the atomic level, i.e. the most detailed level of the data in the data sources, which is a common principle in data warehouse design [58, 53].

Based on the general model shown above, the ground model for the grocery data warehouse is defined as follows. Note that we will use auxiliary functions Sum_p to indicate summation of parameter p in a set of tuples. While it would be possible to define the ASMs without those functions (see Section 4.7), this is done for readability. In TASMs, these auxiliary functions will be replaced by explicit src (structural recursion) constructs.

```
ASM DB-ASM
IMPORT DW-ASM(Shop, Product, Time, Purchase,
  extract_purchase, extract_shop,
  extract_product, extract_time)
EXPORT
  Store, Part, Customer_DB, Buys, Offer
SIGNATURE
  Store(4), Part(3), Customer_DB(4),
  Buys(5), Offer(5),
  r-type(1) (static), req(0) (external)
BODY
  main = if r-type(req) = extract then
    extract_purchase || extract_shop
    extract_product || extract_time
  endif
```

To extract data for the data warehouse relations, for example, the fact table *Purchase*, the following SQL query could be used:

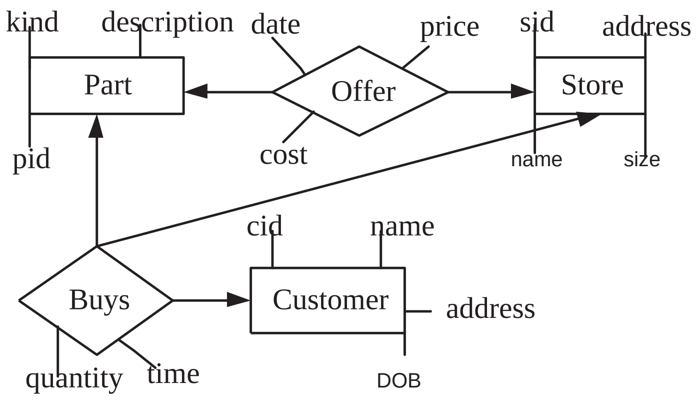

Figure 5.4: The operational database schema

select C.cid, P.pid, S.sid, Date(B.time)
  Sum(B.quantity) **as** quantity,
  Sum(B.quantity) * O.price **as** money_sales,
  Sum(B.quantity) * (O.price - O.cost) **as** profit,
from Part, Store S, Buys B, Offer O, Cutomer_DB C
where Date(B.time) = O.date **and** B.cid = C.cid
and B.pid = P.pid **and** B.sid = S.sid
group by C.cid, P.pid, S.sid, B.time

The DW ASM model is constructed as follows:

```
ASM DW-ASM
IMPORT
  DB-ASM(Store, Part, Buys, Offer),
  OLAP-ASM(V_sales, DM-V_sales, the-datamart, the-matching-view, create_V_sales)
EXPORT
  Shop, Product, Time, Purchase,
  extract_purchase || extract_shop
  extract_product || extract_time
SIGNATURE
  Shop(6), Product(3),
  Time(4), Purchase(7),
  r-type(1) (static), req(0) (external)
BODY
  main = if r-type(req)=open-datamart
    open_datamart(the-datamart(req)) endif

  extract_purchase = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
    $\exists t, q.\text{Buys}(t, i, p, s, q) = 1 \wedge t.date = d \wedge$
    $\exists n, a.\text{Customer}_{db}(i, n, a) \neq \bot \wedge \exists k, d'.\text{Part}(p, k, d') \neq \bot \wedge$
    $\exists a', n', s'.\text{Store}(s, n', s', a') \neq \bot \wedge \text{Offer}(p, s, d, p', c) = 1$
    do let $Q = Sum_q(\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
```

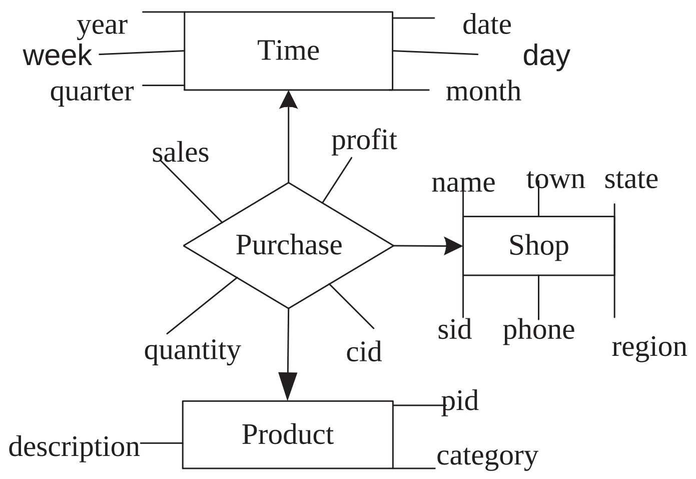

Figure 5.5: The data warehouse schema for sales

```
    $t.date = d\})\ S = Q * p'\ P = Q * (p' - c)$
    in Purchase$(i, p, s, d, Q, S, P) := 1$ enddo

$extract\_shop$ = forall $s, n, a$ with
  $\exists s'$.Store$(s, n, s', a) = 1$
  do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
      in Shop$(s, n, t, r, st, ph) := 1$ enddo

$extract\_product$ = forall $p, k, d$ with
  Part$(p, k, d) \neq \perp$
  do let $p' = p, c = k, d' = d$
      in Product$(p', c, d') := 1$ enddo

$extract\_time$ = forall $t$ with
  $\exists\ c, p, s, q.$ Buys$(c, p, s, q, t) \neq \perp$
  do let $d = t.date, d' = t.day, w = t.week,$
      $q = t.quarter, m = t.month, y = t.year$
      in Time$(d, d', w, q, m, t) := 1$ enddo

$open\_datamart(dm)$ = case the-matching-view$(dm)$ of
  V_sales : $create\_V\_sales$;
            forall $s, r, st, m, q, y, S$ with
              V_sales$(s, r, st, m, q, y, S) = 1$ do
                DM-V_sales$(dm, s, r, st, m, q, y, S) := 1$ enddo
  endcase
```

The following SQL query can be used to create the view of total sales by shop, month and year:

```
select S.sid, S.region, S.state,
       T.month, T.quarter, T.year,
       Sum(P.money_sale) as money_sale
from Shop S, Time T, Purchase P
where P.sid = S.sid and P.date = T.date
group by S.sid, S.region, S.state,
         T.month, T.quarter, T.year
```

The ASM OLAP model is the following.

```
ASM OLAP-ASM
IMPORT DW-ASM(Shop, Product, Time, Purchase)
EXPORT V_sales, DM-V_sales, create_V_sales,
        the-datamart, the-matching-view
SIGNATURE
  V_sales(7) DM-V_sales(8),
  o-type(1),
  owner(1),
  issuer(1),
  the-datamart(1),
  the-view(1),
  the-matching-view(1),
  op(0) (external)
BODY
  main = if o-type(op) = login then LOGIN
      elsif if registered(issuer(op))=1 then
        if o-type(op) = open then OPEN
      elsif o-type(op) = close then CLOSE
      elsif o-type(op) = quit then QUIT
  endif

  LOGIN =
      registered(issuer(op)):=1

  OPEN =
      import dm
        datamart(dm) := 1
        owner(dm) := issuer(op)
        the-matching-view(dm) := the-view(op)
        import reqst
          let reqst = (open-datamart,dm) in
          request(reqst) := 1
        end-import;
      end-import;

  CLOSE =
      close_datamart(the-datamart(op));
```

```
    owner(the-datamart($op$)) := $\perp$
    datamart(the-datamart($op$)) := $\perp$

$QUIT$ =
    let $usr = issuer(op)$ in
       forall $dm$ with owner($dm$) = $usr$
          do $close\_datamart(dm)$
             datamart($dm$) := $\perp$ owner($dm$) := $\perp$ enddo
       registered($usr$) := $\perp$

$close\_datamart(dm)$ = case the-matching-view($dm$) of
 V_sales : forall $s, r, st, m, q, y, S$ with
           DM-V_sales(the-datamart($op$), $s, r, st, m, q, y, S$) = 1 do
             DM-V_sales(the-datamart($op$), $s, r, st, m, q, y, S$) := $\perp$ enddo
 endcase

$create\_V\_sales$ = forall $s, r, st, m, q, y$ with
    $\exists n, t, ph.$Shop$(s, n, t, r, st, ph) = 1 \wedge$
    $\exists d, d', w.$Time$(d, d', w, m, q, y) \neq \perp$
    do let $S = Sum_{s'}$
          $(\{(i, s, p, d, s') \mid \exists q', p'.$
          Purchase$(i, s, p, d, q', s', p') = 1 \wedge$
          $d.month = m \wedge d.year = y\})$
    in V_sales$(s, r, st, m, q, y, S)$ := 1
 enddo
```

5.1.5 The Ground Model in TASM

The database relations in the ASM models above are modelled using boolean functions of fixed arities with no other details. This makes it impossible to compare relations, which is needed in schema integration, nor to use any relational operations, such as projection or join, which are well-used in data intensive applications. In the following, we present the grocery data warehouse in typed ASM (TASM). For simplicity of presentation, we use labels to represent the types with the constraint that they are unique and if two labels are the same, they are of same type too. In TASM, we model database relation using a function R of kind $t \to \{\mathbb{1}\}$, where t is a tuple type $t_1 \times \cdots \times t_n$. We will use the keyword `TASM` instead of `ASM` to indicate that we now use typed ASMs.

```
TASM DB-ASM
IMPORT
  DW-ASM(Shop, Product, Time, Purchase,
  $extract\_purchase$, $extract\_shop$,
  $extract\_product$, $extract\_time$)
EXPORT
  Store, Part, Buys, Offer
SIGNATURE
  Store:$sid \times name \times size \times address \to \{\mathbb{1}\}$,
  Part:$pid \times kind \times description \rightarrow \{\mathbb{1}\}$,
  Customer_DB:$cid \times name \times dob \times address \rightarrow \{\mathbb{1}\}$,
  Buys:$time \times cid \times sid \times pid \times quantity \rightarrow \{\mathbb{1}\}$,
  Offer:$pid \times sid \times date \times price \times cost \rightarrow \{\mathbb{1}\}$,
  r-type:$request \rightarrow rtype$ (static), req:$request$ (external)
BODY
  $main$ = if r-type(req) = $extract$ then
    $extract\_purchase$ || $extract\_shop$
    $extract\_product$ || $extract\_time$
  endif
```

```
TASM DW-ASM
IMPORT
  DB-ASM(Store, Part, Customer_DB, Buys, Offer),
  OLAP-ASM(V_sales, DM-V_sales, the-datamart, the-matching-view, $create\_V\_sales$ )
EXPORT
  Shop, Product, Time, Purchase
  $extract\_purchase$ || $extract\_shop$
  $extract\_product$ || $extract\_time$
SIGNATURE
  Shop:$sid \times name \times town \times region \times state \times phone \rightarrow \{\mathbb{1}\}$,
  Product:$pid \times category \times description \rightarrow \{\mathbb{1}\}$,
  Time:$date \times day \times week \times month \times quarter \times year \rightarrow \{\mathbb{1}\}$,
  Purchase:$cid \times sid \times pid \times date \times qty \times sales \times profit \rightarrow \{\mathbb{1}\}$,
  r-type:$request \rightarrow rtype$ (static), req:$request$ (external)
BODY
  $main$ = if r-type(req) = $open\text{-}datamart$ then
    $open\_datamart(the\text{-}datamart(req))$ endif

  $extract\_purchase$ = forall $i, p, s, d, p', c$ with
    $\exists t.(i, p, s, t, p', c) \in \pi_{cid,pid,sid,time,price,cost}$
        (Buys $\bowtie$ Customer_DB $\bowtie$ Part $\bowtie$ Store $\bowtie$ Offer) $\wedge\, t.date = d$
    do let $Q = src[0, \pi_q, +](\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in Purchase$(i, p, s, d, Q, S, P) := 1$ enddo

  $extract\_shop$ = forall $s, n, a$ with
    $\exists s'.(s, n, s', a) \in$ Store
    do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
        in Shop$(s, n, t, r, st, ph) := 1$ enddo

  $extract\_product$ = forall $p, k, d$ with
    $(p, k, d) \in$ Part
    do let $p' = p, c = k, d' = d$
        in Product$(p', c, d') := 1$ enddo
  $extract\_time$ = forall $t$ with
   $\exists\ c, p, s, q.(c, p, s, q, t) \in$ Buys
   do if Time($t.date, t.day, t.week, t.quarter, t.month, t.year$) = $\perp$
      then Time($t.date, t.day, t.week, t.quarter, t.month, t.year$) := 1
   enddo

  $open\_datamart(dm)$ = case the-matching-view($dm$) of
   V_sales : $create\_V\_sales$;
             forall $s, r, st, m, q, y, S$ with
               $(s, r, st, m, q, y, S) \in$ V_sales do
                 DM-V_sales($dm, s, r, st, m, q, y, S$) := 1 enddo
   endcase

TASM OLAP-ASM
IMPORT DW-ASM(Shop, Product, Time, Purchase)
EXPORT V_sales, DM-V_sales, $create\_V\_sales$, $main$,
        the-datamart, the-matching-view
SIGNATURE
  V_sales:$sid \times region \times state \times month \times quarter \times year \times sales \to \{1\!\!1\}$,
  DM-V_sales:$dm \times sid \times region \times state \times month \times quarter \times$
      $year \times sales \to \{1\!\!1\}$ ,
  o-type: $op \to \{open, close, quit\}$ (static),
  owner: $datamart \to user$,
  issuer: $op \to user$,
  the-datamart: $op \to datamart$,
  the-view: $op \to view$,
  the-matching-view: $datamart \to view$,
  op: $operation$ (external)
BODY
  $main$ = if o-type(op) = $login$ then $LOGIN$
     elsif if registered(issuer(op))=1 then
       if o-type(op) = $open$ then $OPEN$
     elsif o-type(op) = $close$ then $CLOSE$
     elsif o-type(op) = $quit$ then $QUIT$
  endif

  $LOGIN$ =
     registered(issuer(op)):=1

  $OPEN$ =
     import $dm$
       datamart($dm$) := 1
       owner($dm$) := issuer($op$)
       the-matching-view($dm$) := the-view($op$)
       import $reqst$
         let $reqst$ = ($open$-$datamart$,dm) in
         request($reqst$) := 1
        end-import;
    end-import;


$CLOSE$ =
    $close\_datamart$(the-datamart($op$));
    owner(the-datamart($op$)) := $\perp$
    datamart(the-datamart($op$)) := $\perp$


$QUIT$ =
    let $usr = issuer(op)$ in
        forall $dm$ with owner($dm$) = $usr$
            do $close\_datamart(dm)$
                datamart($dm$) := $\perp$ owner($dm$) := $\perp$ enddo
        registered($usr$) := $\perp$


$close\_datamart(dm)$ = case the-matching-view($dm$) of
 V_sales : forall $s, r, st, m, q, y, S$ with
            (the-datamart($op$), $s, r, st, m, q, y, S$) $\in$ DM-V_sales do
              DM-V_sales(the-datamart($op$), $s, r, st, m, q, y, S$) := $\perp$ enddo
 enddo endcase


$create\_V\_sales$ = forall $s, r, st, m, q, y$ with
    $\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
    $\exists d, d', w.(d, d', w, m, q, y) \in$ Time
    do let $S = src[0, \pi_{s'}, +]$
          $(\{(i, s, p, d, s') \mid \exists q', p'.$
          $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
          $d.month = m \wedge d.year = y\})$
    in V_sales($s, r, st, m, q, y, S$) := 1
 enddo
```

5.1.6 Reasoning about the ASM Ground Model

In the ASM method, the ground model serves as the closest translation of the user requirements for the to-be-designed system. The principle of the ASM thesis is to model the system as close as possible to the level of abstraction of the domain, such that users and designers can verify the ground model by inspection. The other advantage of the ground model is that its formalism makes some reasonings on the ground model possible against formalised properties. In the following, examples of reasoning are given for the ground model of grocery store built above.

The ASM ground model developed so far is still rather vague in the sense that it is modelled at rather high level of abstraction. However, it already allows us to start some first correctness checks with respect to the satisfaction of some system properties, such as system invariants. As we adopted the dialogue object approach to data warehouses and OLAP, we will have the following system property:

Lemma 5.1. At any time a datamart is always owned by a user. With respect to the functions we used in the main rule in OLAP-ASM, we can formalise this requirement by

$$\forall dm.\text{datamart}(dm) \neq \perp \Rightarrow \exists usr.\text{owner}(dm) = usr \wedge \text{registered}(usr) \neq \perp \tag{5.1}$$

Proof. The lemma follows from how a datamart is created and maintained:

1. For the initial state, $\forall dm.\text{datamart}(dm) = \perp$, the statement holds.
2. When $\text{datamart}(dm)$ is set to 1 in the "open" rule in OLAP-ASM, the same rule sets $owner(dm) = issuer(op)$. Furthermore, we have the guard $registered(usr) = 1$ in the main rule of OLAP-ASM which ensures that the "open" rule is only called under this condition. Hence, condition (5.1) holds.
3. When $registered(usr)$ is set to $\perp$ in the"quit" rule in OLAP-ASM, the same rule sets $\text{datamart}(dm) = \perp$ for all the dm with $owner(dm) = issuer(op)$, which removes all the datamarts belonging to the user. Thus, condition (5.1) still holds.
4. Ownership of a datamart only changes in three places: In the "open" rule, the owner is set to be a registered user. In the "close" and "quit" rules where ownership of a datamart **dm** is deleted, **datamart(dm)** is also set to $\perp$. Thus condition (5.1) can not get violated this way either.

□

Furthermore, we have system properties regarding the effect of operations such as 'quit' and 'close'. The former closes all the datamarts belonging to the issuer of the operation, the latter deletes a single datamart. We omit formalising these properties using transition constraints. Again, the main rule in OLAP-ASM is consistent with these properties, too.

Finally, considering the consistency of the main rule in OLAP-ASM we have system properties regarding the data content of the data marts, the data warehouse as such and the underlying operational databases. The chosen rules for building datamarts render their realisation as views over the data warehouse explicit.

5.2 The Refinement-based Design Framework

The ASM method assumes that we first set up a ground model. In particular, we have assumed separate ASMs for the database, the data warehouse and the OLAP level. Each of these ASMs uses separate *controlled functions* to model states of the system by logical structures and *rules* expressing transitions between these states. The ASMs are then linked together via queries that are expressed by these transitions. The ground model above captures only the basic requirements, further refinements are required for considering design concerns such as systems optimisation, implementation, and new OLAP requirements. Our design method is developed based on the 3-tier model and the ASM method. The former provides the logical structure of the system, the latter provides the step-by-step refinement approach. In the following, we group refinements under three categories: requirements capture, system optimisation, and system implementation. For each category we present a set of refinement rules, in a rather abstract manner, that are ultimately decomposed and formalised into a set of concrete rules in a later development stage, for example, the formal rules for view integration in Chapter 6.

5.2.1 Requirements Capture

Like most software systems, our design method for data warehouse design begins with the requirements from the user end, i.e. the OLAP system. We build data warehouse schemas based on what OLAP needs, for example, the set of analysis queries or reports. As this is not a one-off process due to the dynamic nature of business analysis, it is not uncommon that we may need to deal with new OLAP requirements regularly after the data warehouse has been implemented. In particular, our data warehouse design starts from a data mart for a single business process, more data marts are to be added later on. The new requirements may require changes in the data warehouse schemata, such as adding new schemata. This type of changes can result in inconsistency in the data warehouse if schema integration is not considered. We tackle this problem with the schema integration technique, i.e. we integrate the set of new data schema from the new requirements with the existing data warehouse schema to resolve the inconsistency, and at the same time we maintain a data warehouse with little redundancy. Schema integration will be dealt with in Chapter 6. New requirements can sometimes be presented in the form of other than purely new queries, such as modified OLAP queries, which we will not deal with here. Our focus here is on the incremental design of data warehouse.

Using the data warehouse/OLAP ASM ground model as a basis, we handle new OLAP requirement, such as adding new OLAP queries, as follows: in the OLAP ASM, we define the new OLAP functions and rules. In the data warehouse ASM, we define new functions and rules, and modify the existing functions and rules to support the new OLAP queries. In the database ASM, we add and change functions and rules to support the changes of the data warehouse. In fact, we are propagating the changes gradually from the OLAP tier down to the data warehouse tier and then to the operational database tier.

The refinements for requirement capturing are classified under conservative extension or incremental refinement in the ASM method. That means, the existing functions will be preserved when new features are added in the refinement. We build a refinement process for systematically capturing new requirements in data warehouse and OLAP system design as follows.

1. *Add a new rule to the OLAP ASM*: this is used to model an additional OLAP function by adding a new rule name and the definition of the rule for the new function to the OLAP ASM.

   Discussion: *it is presumed that the newly added functionality is not present in the OLAP ASM before. The new rule will work under a condition such that it is not defined for the old machine. In such case, adding a new rule preserves the existing functionality of the old machine. When a new rule or function is defined, we need to decide if it should be export or import for reference.*

2. *Add a new controlled function to the OLAP ASM*: this is used to model a view that is needed for the support of any new OLAP function, provided the existing view definitions are not yet sufficient.

3. *Add new controlled function(s) to the DW ASM*: this is used to model the schema that is needed in supporting the new OLAP function in data warehouse tier.

4. *Integrate controlled functions on the DW ASM*: this is used whenever the schema is extended. As a consequence, the view creation rules on the OLAP ASM must be

changed accordingly.

Discussion: *the integration process aims to preserve the information by the notion of schema equivalence and dominance when two schemas are integrated. This step relates to a set of schema transformation rules which is discussed in Chapter 6.*

5. *Add controlled functions to the DB ASM*: this is used to add new schema to support DW ASM, provided the existing data is not sufficient.
6. *Integrate controlled functions on the DB ASM*: this is used whenever the schema is extended. As a consequence, the extraction rules on the DW ASM must be changed accordingly.
7. *Change the rules on DW ASM*: this is used to adapt the rules defined for extracting data for the data warehouse relations to the changes in the data warehouse schema. This is again an impact from schema integration.
8. *Change the rules on DB ASM*: this is used to adapt the rules are used in data extraction upon data warehouse refresh request. Any changes, either new addition or updates, to data extraction rules should be reflected in the related rules in DB ASM.
9. *Change the functions/rules on OLAP ASM*: this is used to change the functions or rules that are affected in this refinement process, such as rules that make reference to the schemas which are changed during the integration, or rules that process the newly added OLAP functions.

5.2.2 Optimisation

Some refinements are used to optimise the performance of the system. These refinement rules are applied to reorganise the specification independently from the user requirements. Refinements for system optimisation can be classified under procedural or data refinement in the ASM method. To be tailored for data warehouse and OLAP system design, some typical optimisation steps are considered:

1) To materialise the OLAP views. That is, to compute the OLAP queries in advance and store them as views in the data warehouse. When the queries are called, they can be answered by the stored views instantly from the data warehouse without waiting for computation of the queries. This will speed up the system performance particularly as business analysis is usually data intensive, but it also results in the issue of view maintenance, which we will discuss further later on in Chapter 7.

2) To update the data warehouse incrementally. That is, not to recompute the queries from scratch, as is the case in our ground model but only propagate the changes to the data warehouse.

The tailored refinement process for systematically incorporating the above two optimisation steps is specified as follows:

10. *Incorporate view materialisation* :
    (a) *Add new controlled function in DW ASM* : this is used to add the OLAP views to the data warehouse as the materialised views.

Discussion: *for an effective approach in view selection we can adopt some selection process or algorithm such as the one we used in the case study later in Chapter 7.*

(b) *Integrate materialised views in DW ASM* : this is used to reduce redundancy that may have occurred after more views are materialised. A set of transformation rules can be applied, which will be discussed in detail in Chapter 6.

(c) *Add new rules in DW ASM* : this is used to define the transition rules to maintain the materialised views up to date with the data warehouse changes. These rules are called after each refreshing of the data warehouse.

(d) *Change the rules in DW ASM* : these rules are for opening a datamart for the OLAP ASM. After view materialisation or view integration, these rules need to be adapted too.

11. *Incorporate incremental updates* :

    (a) *Add monitored functions in DB ASM* : this is used to define relations to store updates of the source relations, called delta files.

    (b) *Add controlled functions in DW ASM* : this is used to define relations to store computed changes for data warehouse relations.

    (c) *Add rules in DW ASM* : this is used to define the rules for computing the changes from source relations and propagating changes into data warehouse relations.

    (d) *Replace rules in DB ASM* : this is used to replace the refresh rules with the rules for incremental updates.

5.2.3 Implementation

The final group of the refinements in our discussion is the system implementation refinements. Refinements for system implementation can be classified under procedural or data refinement in the ASM method. The refinements introduced in the following are mainly designed for realising high-level design decisions such as data distribution and incremental maintenance. This group of refinements are classified under the procedural or data refinement in the ASM method.

12. *Apply implementation refinements*: these refinement rules apply to the ASMs on all three levels and consist of realising design decisions for moving the ASMs closer to their implementation while preserving the semantics of runs. It is not our focus here to discuss how to move specification to codes, which is thoroughly studied, for example in [123, 78, 26], and particularly for data intensive applications in [97].

13. *Distribution design*: as it is common that an enterprise is geographically distributed, data warehouse design methods should take distribution design into consideration. More detailed discussion on distribution design is shown in Chapter 7. Our distribution design concerns the database instead of communication design. We use nodes to describe the locations where a local data warehouse resides.

    (a) *Replicate the data warehouse and the OLAP ASMs*: for each node in the network assume the same copy of the data warehouse ASM and the OLAP ASM.

(b) *Remove controlled functions and rules in local OLAP ASMs*: if the needed OLAP functionality is different at different network nodes, then these rules will simply reduce the corresponding OLAP ASM.

(c) *Fragment controlled functions in local data warehouse ASMs*: these rules will reorganise and reduce a local data warehouse ASM, if the corresponding OLAP ASM does not need all of the replicated data warehouse. The refresh rules are then adapted accordingly.

(d) *Recombine fragments in local data warehouse ASMs*: these rules will reorganise a local data warehouse ASM according to query cost considerations. The refresh rules are then adapted accordingly.

(e) *Adapt the view creation rules accordingly in local OLAP ASM*: this is used when fragmentation is implemented. The OLAP views are created over the fragments at the local data warehouse.

5.3 Some Refinements

In the above, we have constructed a data warehouse ground model and developed a design framework for data warehouse and OLAP system development. In the following, we will use the grocery store as an example to present some refinements in the areas of capturing new requirements and system optimisation. The distribution design will be presented in Chapter 7.

5.3.1 Incorporating Roll-up and Drill-down

Roll-up and drill-down are two typical OLAP functionalities which change the granularity of current aggregations. In the following, we use the grocery store model as an example to show how roll-up and drill-down can be incorporated into the model using the refinement-based method.

Example 5.1. In the grocery store ground model, we have created a view on sales by shop, month, and year at the OLAP ASM. A roll-up operation can be used to move the aggregation on sales from shop to region to state, or from month to quarter to year, that is, from low to high along the dimensions **Location** or **Time**. A drill-down operation does just the opposite, from high to low granularity.

To add the *roll-up* and *drill-down*, we proceed as follows:

1. We first apply rule #1 (*Add a new rule to the OLAP ASM*) to add transition rules *roll-up*(dm, to) and *drill-down*(dm, to) on the OLAP ASM, which move the aggregation from the current level shown in the datamart dm to the level indicated by to.

2. To support the new operations, we apply rule #2 (*Add a new controlled function to the OLAP ASM*) to add a new function **up-to**:*operation* $\rightarrow$ *level*, with *level* as a new universe or a new type in the typed ASM. This function maps the operation *roll-up* or *drill-down* to the level where the aggregation will be performed.

3. The operation *roll-up* or *drill-down* only needs to refer back to the original view on which the datamart is based, as aggregation from original is more straightforward. Since it does not result in changes in data requirement, no application of the rules for schema integration or re-organisation is required.

4. To incorporate *roll-up* and *drill-down* as new operations, the function **type** and the rule *main* need to be adapted by creating new conditions for the new functions to be executed. We apply rule #8 (*Change the functions/rules on OLAP ASM*) resulting in the following:

```
o-type: $op \rightarrow$ {login, open, close, quit, roll-up, drill-down }

$main$ = if o-type($op$) = $login$ then $LOGIN$
    elsif o-type($op$) = $open$ then $OPEN$
    elsif o-type($op$) = $close$ then $CLOSE$
    elsif o-type($op$) = $quit$ then $QUIT$
  elsif o-type($op$) = $roll\text{-}up$ then $ROLL\text{-}UP$
  elsif o-type($op$) = $roll\text{-}up$ then $DRILL\text{-}DOWN$
endif

$ROLL\text{-}UP$ = $roll\text{-}up(the\text{-}datamart(op), up\text{-}to(op))$
$DRILL\text{-}DOWN$ = $drill\text{-}down(the\text{-}datamart(op), up\text{-}to(op))$
```

There are many ways to implement the operations *roll-up* and *drill-down.* As an example, we may simply re-aggregate from bottom to *up-to*(*op*) using the view that the datamart is based on. Whether is for *roll-up* or *drill-down*, we may use this aggregation rule to aggregate up to the level indicated by *up-to*(*op*).

```
$roll\text{-}up(dm, to) = aggr(dm, to)$
$drill\text{-}down(dm, to) = aggr(dm, to)$

$aggr(dm, to)$ = case the-matching-view($dm$) of
  V_sales :
        if $to$ = shop then
          forall $s, r, st, m, q, y, S$ with
            $(s, r, st, m, q, y, S) \in$ V_sales do
              DM-V_sales$(dm, s, r, st, m, q, y, S) := 1$ enddo
        elsif $to$ = region then
          forall $r, st, m, q, y$ with
            $\exists s, S.(s, r, st, m, q, y, S) \in$ V_sales do let
              $S' = src[0, \pi_S, +]$ $(\{(s, S) \mid$
                $(s, r, st, m, q, y, S) \in$ V_sales
              in DM-V_sales$(dm, \perp, r, st, m, q, y, S') := 1$ enddo
        elsif $to$ = state then
          forall $st, m, q, y$ with
            $\exists s, r, S, P.(s, r, st, m, q, y, S) \in$ V_sales do let
              $S' = src[0, \pi_S, +]$ $(\{(s, r, S) \mid$
                $(s, r, st, m, q, y, S) \in$ V_sales
             in DM-V_sales$(dm, \perp, \perp, st, m, q, y, S') := 1$ enddo
        ...
  endcase
```

We have omitted the aggregation on the dimension **Time** as it is similar to what we presented for the dimension **Location** above.

□

The above refinement extends the functionality in the old machine. What we need to ensure is that the refinement preserves the functionality in the old machine.

Lemma 5.2. The above refinement preserves functions in the old machine.

Proof. It follows by verifying the conditions for rule applications in the machines.

- The input in both machines is *op*, which consists of the *type* of operation, the *issuer*, the *datamart* if needed, and the corresponding *view*.
- If o-type(*op*) maps to "login" "open", "quit", or "close", both machines perform the same actions;
- If o-type(*op*) maps to "roll-up" or "drill-down", the old machine does nothing as the rules are not defined, the refined machine acts as defined.

□

5.3.2 Materialising OLAP Views

As OLAP queries are data intensive, performance is always a concern in data warehouse design. Pre-computing the queries and storing the result in the data warehouse becomes one of the design options to overcome the performance problem. What views to materialise, i.e. the view selection problem, is not our focus here. Instead we model the corresponding changes in the data warehouse process when view materialisation is incorporated. In the following we apply the refinement step using the grocery store as an example.

For each OLAP view to be materialised, we first define a function for storing the materialised view at the data warehouse, and then perform view integration on the materialised views for reducing unnecessary redundancy. We add new transition rules for newly added materialised views if any. Those rules are called whenever the data warehouse is refreshed, to recompute the query over the current data warehouse. Finally, all the transition rules in the three modules are adapted if views involved in the rules are affected in the view integration.

Example 5.2. As our grocery store ground model supports one single OLAP view, the changes in the view materialisation refinement do not involve view integration as shown in the following:

```
TASM DB-ASM
IMPORT
  DW-ASM(..., MV_V_sales, refresh-MV_V_sales)
  ...
BODY
  main = if r-type(req) = extract then
    (extract_purchase || extract_customer || extract_shop
    extract_product || extract_time); refresh-MV_V_sales
  endif
```

```
TASM DW-ASM
IMPORT
  ...
EXPORT
  ...
SIGNATURE
  ...
  MV_V_sales:$sid \times region \times state \times month \times quarter \times year \times$
       $sales \to \{\mathbb{1}\}$

BODY
  ...

  $refresh\text{-}MV\_V\_sales$ =
       $create\_V\_sales$;
       forall $s, r, st, m, q, y, S$ with
          $(s, r, st, m, q, y, S) \in$ V_sales do
             MV_V_sales$(s, r, st, m, q, y, S) := 1$ enddo

  $open\_datamart(dm)$ = case the-matching-view$(dm)$ of
   V_sales : forall $s, r, st, m, q, y, S$ with
               $(s, r, st, m, q, y, S) \in$ MV_V_sales do
                  DM-V_sales$(dm, s, r, st, m, q, y, S) := 1$ enddo
   endcase

  ...
```

In the above, only the modified functions and rules in the refined model are presented. The model shows that incorporating view materialisation in DW ASM is an internal change and has no effects that are noticeable to the users. A more complex example on view materialisation which involves view selection and view integration will be discussed as a case study in Chapter 7.

□

As our major concern in computing data marts over the materialised views is that the data in materialised views is consistent with the data in the data warehouse. We denote this as ins(MV) $\simeq$ ins(DW), meaning that the instances for the materialised views are the results of executing their queries over the data warehouse instance.

Lemma 5.3. OLAP datamarts when opened, are consistent with the data warehouse in both machines.

Proof. It follows from run induction.

- When r-type(req)=*extract*: after the rule application, we have ins(MV) $\simeq$ ins(DW) in the refined machine, meaning the instance of materialised views is consistent with the instance of the data warehouse.
- When r-type(req)=*open-datamart*: after applying the rule *open_datamart*, $dm \simeq$ ins(DW) holds in the old machine, $dm \simeq$ ins(MV), in the refined machine. While

the data warehouse is under refreshing (this is done in a single ASM step), no OLAP requests will be answered. Thus ins(MV) $\simeq$ ins(DW) holds until next application of *extract*. Hence, by the meaning of consistency, we have $dm \simeq$ ins(DW) in the refined machine.

$\square$

5.3.3 Incremental Updates

In our ground model, the data warehouse is maintained by recomputing the queries from scratch upon extraction request. Often these computations are data-intensive and thus rather expensive. As an alternative approach for view maintenance, incremental update has been suggested by which the data warehouse can be maintained by propagating only the changes in the source data into the data warehouse algorithmically (e.g. [38, 36, 87]). Incremental updates have been modelled as a refinement in our refinement process for data warehouse design. The purpose is to provide a systematic way for adapting the process when incremental update is incorporated.

As our focus here is not to design an algorithm for incremental updates but to show how incremental updates can be incorporated into the data warehouse process, we adopt the basic ideas in the approaches suggested in [38, 36, 87], that is, the changes from the operational database are captured in some delta files, which then are used to compute the changes to the data warehouse, and finally computed changes are propagated into the data warehouse. In order to avoid getting involved too much in the implementation details, we presume that in the operational applications there is a mechanism for capturing the changes to the delta files. Furthermore, we assume that data items which used to be valid are not to be deleted. For example, closing down a store will not result in deletion of the store, although the store may be indicated as obsolete in the operational applications. This assumption will allow us to keep the historical data in the data warehouse.

Another issue related to incremental updates is how to process summarised data, especially when it is not shown explicitly, as is the case for **Time** in our grocery store example. The function **Time** is populated using **Buys** in such a way that there may be multiple tuples in **Buys** resulting in the same tuple to be added to **Time**. As we only keep one of them, it is not straightforward to process the deletions in Δ-Buys_del, that is, you cannot decide if the tuples corresponding to the tuples in Δ-Buys_del should be deleted from **Time** or not, because you cannot decide if this is the only tuple which derives the tuple in **Time**. This problem has been addressed in the work of incremental updates by using bags instead of sets (e.g. [87]). As stated at the beginning, the update algorithm is not the focus here, so we will not consider incremental update in relations such as **Time**, although the way **Time** is populated is not efficient.

Example 5.3. Let us again use the grocery store as an example to show how the ground model is adapted when incremental updates are incorporated.

1. We first define the functions in the DB ASM for capturing the changes as inserted or deleted, for updates are modelled as delete-insert pair, since the last refresh from the source relations separately.

   Δ-Buys_ins:$time \times cid \times sid \times pid \times quantity \rightarrow \{\mathbb{1}\}$,
   Δ-Store_ins:$sid \times name \times size \times address \rightarrow \{\mathbb{1}\}$,
   Δ-Part_ins:$pid \times kind \times descriptin \rightarrow \{\mathbb{1}\}$,
   Δ-Customer_DB_ins:$cid \times name \times dob \times address \rightarrow \{\mathbb{1}\}$,

Δ-Offer_ins:$pid \times sid \times date \times price \times cost \rightarrow \{\mathbb{1}\}$,
Δ-Buys_del:$time \times cid \times sid \times pid \times quantity \rightarrow \{\mathbb{1}\}$,
Δ-Store_del:$sid \times name \times size \times address \rightarrow \{\mathbb{1}\}$,
Δ-Part_del:$pid \times kind \times descriptin \rightarrow \{\mathbb{1}\}$,
Δ-Customer_DB_del:$cid \times name \times dob \times address \rightarrow \{\mathbb{1}\}$,
Δ-Offer_del:$pid \times sid \times date \times price \times cost \rightarrow \{\mathbb{1}\}$,

2. We take a progressive approach to incorporate the changes from the data sources. For each of the relations Buys, Customer_DB, Part, Store and Offer, we define a controlled function in DW ASM for capturing the computed changes in **Purchase** after previous updates have been completed.

   Δ-Purchase_B_ins:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_C_ins:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_P_ins:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_S_ins:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_O_ins:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_B_del:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_C_del:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_P_del:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_S_del:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$
   Δ-Purchase_O_del:$cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{\mathbb{1}\}$

3. We define the following transition rules for computing the changes in **Purchase** in DW ASM, for ease of reading, we define the following short forms:

   Buys$_{old}$ = Buys ∪ Δ-Buys_del − Δ-Buys_ins
   Store$_{old}$ = Store ∪ Δ-Store_del − Δ-Store_ins
   Part$_{old}$ = Part ∪ Δ-Part_del − Δ-Part_ins
   Customer_DB$_{old}$ = Customer_DB ∪ Δ-Customer_DB_del − Δ-Customer_DB_ins
   Offer$_{old}$ = Offer ∪ Δ-Offer_del − Δ-Offer_ins

```
compute_buys_ins = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
 $\exists t.(i, p, s, t, p', c) \in \pi_{cid,pid,sid,time,price,cost}$
    (Δ-Buys_ins ⋈ Customer_DB$_{old}$ ⋈ Part$_{old}$ ⋈ Store$_{old}$ ⋈ Offer$_{old}$)
       $\wedge t.date = d$
 do let $Q = src[0, \pi_q, +](\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
    $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
     in Δ-Purchase_B_ins$(i, p, s, d, Q, S, P) := 1$ enddo

compute_customer_ins = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
 $\exists t.(i, p, s, t, p', c) \in \pi_{cid,pid,sid,time,price,cost}$
    (Buys ⋈ Δ-Customer_DB_ins ⋈ Part$_{old}$ ⋈ Store$_{old}$ ⋈ Offer$_{old}$)
       $\wedge t.date = d$
 do let $Q = src[0, \pi_q, +](\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
    $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
```

```
        in $\Delta$-Purchase_C_ins$(i,p,s,d,Q,S,P) := 1$ enddo

    $compute\_part\_ins$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
     $\exists t.(i,p,s,t,p',c) \in \pi_{cid,pid,sid,time,price,cost}$
        (Buys $\bowtie$ Customer_DB $\bowtie$ $\Delta$-Part_ins $\bowtie$ Store$_{old}$ $\bowtie$ Offer$_{old}$)
          $\wedge t.date = d$
     do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\Delta$-Purchase_P_ins$(i,p,s,d,Q,S,P) := 1$ enddo

    $compute\_store\_ins$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
     $\exists t.(i,p,s,t,p',c) \in \pi_{cid,pid,sid,time,price,cost}$
        (Buys $\bowtie$ $\Delta$-Customer_DB_ins $\bowtie$ Part$_{old}$ $\bowtie$ Store$_{old}$ $\bowtie$ Offer$_{old}$)
          $\wedge t.date = d$
     do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\Delta$-Purchase_S_ins$(i,p,s,d,Q,S,P) := 1$ enddo

    $compute\_offer\_ins$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
     $\exists t.(i,p,s,t,p',c) \in \pi_{cid,pid,sid,time,price,cost}$
        (Buys $\bowtie$ Customer_DB $\bowtie$ Part $\bowtie$ Store $\bowtie$ $\Delta$-Offer_ins)
          $\wedge t.date = d$
     do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\Delta$-Purchase_O_ins$(i,p,s,d,Q,S,P) := 1$ enddo

    $compute\_buys\_del$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
     $\exists t.(i,p,s,t,p',c) \in \pi_{cid,pid,sid,time,price,cost}$
        ($\Delta$-Buys_ins $\bowtie$ Customer_DB$_{old}$ $\bowtie$ Part$_{old}$ $\bowtie$ Store$_{old}$ $\bowtie$ Offer$_{old}$)
          $\wedge t.date = d$
     do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\Delta$-Purchase_B_del$(i,p,s,d,Q,S,P) := 1$ enddo

 ...

    $compute\_offer\_del$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
     $\exists t.(i,p,s,t,p',c) \in \pi_{cid,pid,sid,time,price,cost}$
        (Buys $\bowtie$ Customer_DB $\bowtie$ Part $\bowtie$ Store $\bowtie$ $\Delta$-Offer_del)
          $\wedge t.date = d$
     do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\Delta$-Purchase_O_del$(i,p,s,d,Q,S,P) := 1$ enddo
```

We have omitted the details in computing changes using deletions as the process is the same as for the inserts.

4. Then we define rules for propagating changes into the data warehouse in DW ASM:

```
$prop\_purchase\_B\_ins =$ forall $i, p, s, d, Q, S, P$ with
    $(i, p, s, d, Q, S, P) \in \Delta$-Purchase_B_ins
      do if $\exists Q', S', P'.(i, p, s, d, Q', S', P') \in$ Purchase
        then let $(Q', S', P')$.
          $\mathbf{I}Q', S', P'.((i, p, s, d, Q', S', P') \in$ Purchase$)$ in
          Purchase$(i, p, s, d, Q', S', P') := \perp$,
          Purchase$(i, p, s, d, Q + Q', S + S', P + P') := 1$
        else Purchase$(i, p, s, d, Q, S, P) := 1$
      enddo
```

prop_purchase_C_ins$=$ ...

...

```
$prop\_purchase\_B\_del =$ forall $i, p, s, d, Q, S, P$ with
    $(i, p, s, d, Q, S, P) \in \Delta$-Purchase_B_del
      do if $\exists Q', S', P'.(i, p, s, d, Q', S', P') \in$ Purchase
        then do let $(Q', S', P')$.
          $\mathbf{I}Q', S', P'.((i, p, s, d, Q', S', P') \in$ Purchase$)$ in
          Purchase$(i, p, s, d, Q', S', P') := \perp$,
          if $\neg(Q = Q' \wedge S = S' \wedge P = P')$ then
            Purchase$(i, p, s, d, Q - Q', S - S', P - P') := 1$ enddo
      enddo
```

...

```
$prop\_shop\_ins =$ forall $s, n, a$ with
 $\exists s'.(s, n, s', a) \in \Delta$-Store_ins
 do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
     in Shop$(s, n, t, r, st, ph) := 1$ enddo

$prop\_product\_ins =$ forall $p, k, d$ with
 $(p, k, d) \in \Delta$-Part_ins
 do let $p' = p, c = k, d' = d$
     in Product$(p', c, d') := 1$ enddo

$prop\_shop\_del =$ forall $s, n, a$ with
 $\exists s'.(s, n, s', a) \in \Delta$-Store_del
 do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
     in Shop$(s, n, t, r, st, ph) := \perp$ enddo

$prop\_product\_del =$ forall $p, k, d$ with
 $(p, k, d) \in \Delta$-Part_del
 do let $p' = p, c = k, d' = d$
     in Product$(p', c, d') := \perp$ enddo
```

5. We define refresh rules in DW ASM:

```
$refresh\_purchase =$
  $(compute\_buys\_ins \parallel compute\_buys\_del$
  $compute\_customer\_ins \parallel compute\_customer\_del$
  $compute\_store\_ins \parallel compute\_store\_del$
  $compute\_part\_ins \parallel compute\_part\_del$
  $compute\_offer\_ins \parallel compute\_offer\_del);$
  $prop\_purchase\_B\_ins; prop\_purchase\_B\_del;$
  $prop\_purchase\_C\_ins; prop\_purchase\_C\_del;$
  $prop\_purchase\_S\_ins; prop\_purchase\_S\_del;$
  $prop\_purchase\_P\_ins; prop\_purchase\_P\_del;$
  $prop\_purchase\_O\_ins; prop\_purchase\_O\_del$
$refresh\_shop =$
  $prop\_shop\_ins; prop\_shop\_del$
$refresh\_product =$
  $prop\_product\_ins; prop\_product\_del$
```

6. Finally we adapt the changes into the rules for incremental updates in DB ASM:

```
$main$ = if r-type(req) = $extract$ then
    $extract\_purchase \parallel extract\_shop$
    $extract\_product \parallel extract\_time$
  elsif r-type(req) = $refresh$ then
    $refresh\_purchase \parallel refresh\_shop$
    $refresh\_product \parallel extract\_time$   endif
```

□

When incremental update is incorporated as an extended functionality, as the case above, it is easy to check the refinement is correct since it preserves the existing functionalities in the old machine. Thus it would be more meaningful to show that the two sets of rules, one for *extract*, one for *refresh* have the same effects.

Lemma 5.4. *refresh* works the same as *extract*.

Proof. It requires to verify if the computations of the data warehouse relations are the same by the two sets of rules.

- It is easy to verify by going through the rules that the above is the case for **Shop**, **Product**, and **Time**.
- For **Purchase**, the proof is a bit complicated. It involves join operations among a number of source relations and projections and aggregations, though the later two operations are rather simple. Thus the issue is how changes can be correctly propagated through the join operations. While the computation of **Purchase** involves joining **Buys, Customer_DB, Part, Store** and **Offer**, among other operations, we will just use a simple example to give a sketch of the proof here.

  Let us assume $A = B \bowtie C \bowtie D$. Following the method for *refresh* as above, the changes of A is computed incrementally based on the changes of B, C, and D as follows:

- $\Delta A_B := \Delta B \bowtie C_{old} \bowtie D_{old}$
- $\Delta A_C := B_{new} \bowtie \Delta C \bowtie D_{old}$
- $\Delta A_D := B_{new} \bowtie C_{new} \bowtie \Delta D$
- $\Delta A := \Delta A_B \cup \Delta A_C \cup \Delta A_D$.
- $A_{new} := A_{old} \cup \Delta A$.

So what we need to prove is that the following holds:

$$A_{new} = B_{new} \bowtie C_{new} \bowtie D_{new}$$

All we need here is that the join $\bowtie$ operator is commutative and associative, and interacts with union $\cup$ in a distributive manner, that is

$$R_1 \bowtie (R_2 \cup R_3) = (R_1 \bowtie R_2) \cup (R_1 \bowtie R_3)$$

We proceed as follows:

$$\begin{aligned}
A_{new} &= A_{old} \cup \Delta A \\
&= A_{old} \cup \Delta A_B \cup \Delta A_C \cup \Delta A_D \\
&= B_{old} \bowtie C_{old} \bowtie D_{old} \cup \Delta B \bowtie C_{old} \bowtie D_{old} \cup \Delta A_C \cup \Delta A_D \\
&= (B_{old} \cup \Delta B) \bowtie C_{old} \bowtie D_{old} \cup \Delta A_C \cup \Delta A_D \\
&= B_{new} \bowtie C_{old} \bowtie D_{old} \cup B_{new} \bowtie \Delta C \bowtie D_{old} \cup \Delta A_D \\
&= B_{new} \bowtie (C_{old} \cup \Delta C) \bowtie D_{old} \cup \Delta A_D \\
&= B_{new} \bowtie C_{new} \bowtie D_{old} \cup B_{new} \bowtie C_{new} \bowtie \Delta D \\
&= B_{new} \bowtie C_{new} \bowtie (D_{old} \cup \Delta D) \\
&= B_{new} \bowtie C_{new} \bowtie D_{new}
\end{aligned}$$

The proof for deletions can be constructed in the same fashion.

□

Chapter 6

View Integration

Our data warehouse design method is an incremental design method, which resembles the bottom-up approach in [58], but is different from it in that the issue of system integrity and data consistency is resolved by the schema integration technique. We have shown in Chapter 5 how this technique is integrated in the refinement-based design framework.

Our schema integration is based on HERM (Higher-order Entity-Relationship Model), an extended ER model. A review in this area is presented in Chapter 2. The set of schema transformation rules presented in the following is based on the ones in [75], with small modifications which are discussed in Section 6.5.

We start with the introduction of the basics of HERM in Section 6.1, and the query language for HERM in Section 6.2, followed by the notion of schema equivalence and dominance in Section 6.3. A pragmatic method for schema integration is given in Section 6.4. Finally, the focus of this chapter, the transformation rules of schema integration and the corresponding formal rules in TASM are presented in Section 6.5.

6.1 HERM

As we will base our presentation on the higher-order Entity-Relationship model (HERM) [106], we start with a brief review of the model as far as it is important for our purposes here. In particular, we focus on algebraic and logical query languages for HERM. These will be needed to address the important issue of defining schema dominance and equivalence in a way that the expressiveness is sufficient for the integration of extended views as needed for data warehouses. The basic case dealing only with plain views over HERM schemata, i.e. ignoring the extensions by operations, adaptivity, etc. was already handled in [66, 67].

The major extensions of the HERM compared with the flat ER model concern nested attribute structures, higher-order relationship types and clusters, a sophisticated language of integrity constraints, operations, dialogues and their embedding in development methods. Here we only review some of the structural aspects.

In the following let $\mathcal{A}$ denote some set of *simple attributes*. Each simple attribute $A \in \mathcal{A}$ is associated with a base domain $dom(A)$, which is some fixed countable set of values. The values themselves are of no particular interest.

In HERM it is permitted to define nested attributes. For this take a set $\mathcal{L}$ of *labels* with the only restriction that labels must be different from simple attributes, i.e. $\mathcal{L} \cap \mathcal{A} = \emptyset$.

Definition 6.1. A *nested attribute* is either a simple attribute, the null attribute $\perp$, a tuple attribute $X(A_1, \dots, A_n)$ with pairwise different nested attributes A_i and a label

$X \in \mathcal{L}$ or a set attribute $X\{A\}$ with a nested attribute A and a label $X \in \mathcal{L}$. Let $\mathcal{NA}$ denote the set of all nested attributes.

We extend *dom* to nested attributes in the standard way, i.e. a tuple attribute will be associated with a tuple type, a set attribute with a set type, and the null attribute with $dom(\perp) = \mathbb{1}$, where $\mathbb{1}$ is the trivial domain with only one value.

In principle we could also permit other constructors than tuple and set constructors, e.g. constructors $\langle\cdot\rangle$ and $[\cdot]$ for multisets and lists. This would, however, only complicate our presentation here without leading to additional insights. We therefore disregard these other constructors.

On nested attributes we have a partial order $\geq$ defined as follows.

Definition 6.2. $\geq$ is the smallest partial order on $\mathcal{NA}$ with

- $A \geq \perp$ for all $A \in \mathcal{NA}$,
- $X\{A\} \geq X\{A'\} \Leftrightarrow A \geq A'$ and
- $X(A_1, \dots, A_n) \geq X(A'_1, \dots, A'_m) \Leftrightarrow \bigwedge_{1 \leq i \leq m} A_i \geq A'_i$.

A *generalised subset* of a set $F \subseteq \mathcal{NA}$ of nested attributes is a set $G \subseteq \mathcal{NA}$ of nested attributes such that for each $A' \in G$ there is some $A \in F$ with $A \geq A'$.

It is easy to see that $X \geq X'$ gives rise to a canonical projection $\pi^X_{X'} : dom(X) \to dom(X')$.

Let us now define the entity and relationship types and clusters in HERM using the following compact definition.

Definition 6.3. A *level-0-type* E consists of a set $attr(R) = \{A_1, \dots, A_m\}$ of nested attributes and a key $key(R)$. A *level-k-type* R ($k > 0$) consists of a set $comp(R) = \{r_1 : R_1, \dots, r_n : R_n\}$ of labelled components with pairwise different labels r_i, a set $attr(R) = \{A_1, \dots, A_m\}$ of nested attributes and a key $key(R)$. Each component R_i is a type or cluster of a level at most $k-1$, but at least one of the R_i must be level-$(k-1)$-type or -cluster.

For the key we have $key(R) = comp'(R) \cup attr'(R)$ with $comp'(R) \subseteq comp(R)$ and a generalised subset $attr'(R)$ of the set of attributes.

A *level-k-cluster* is $C = R_1 \oplus \dots \oplus R_n$ with pairwise different components R_i, each of which is a type or cluster of a level at most k. At least one of the R_i must be level-k-type, or -cluster.

The labels r_i used in components are called *roles*. Roles can be omitted in case the components are pairwise different. A level-0-type E is usually called an *entity type*, a level-k-type R with $k > 0$ is called a *relationship type*.

A *HERM schema* is a finite set $\mathcal{S}$ of entity types, relationship types and clusters together with a set Σ of integrity constraints defined on $\mathcal{S}$. We write $(\mathcal{S}, \Sigma)$ for a schema, or simply $\mathcal{S}$, if $\Sigma = \emptyset$.

Note that the notion of level has only been introduced to exclude cycles in schemata. Besides this it has no other meaning. Conversely, if there are no cycles in a schema, then there is a straightforward way to assign levels to the types and clusters in the schema such that the conditions in Definition 6.3 are satisfied.

Example 6.1. The following type definitions define a HERM schema for a loan application as it might be used by some bank:

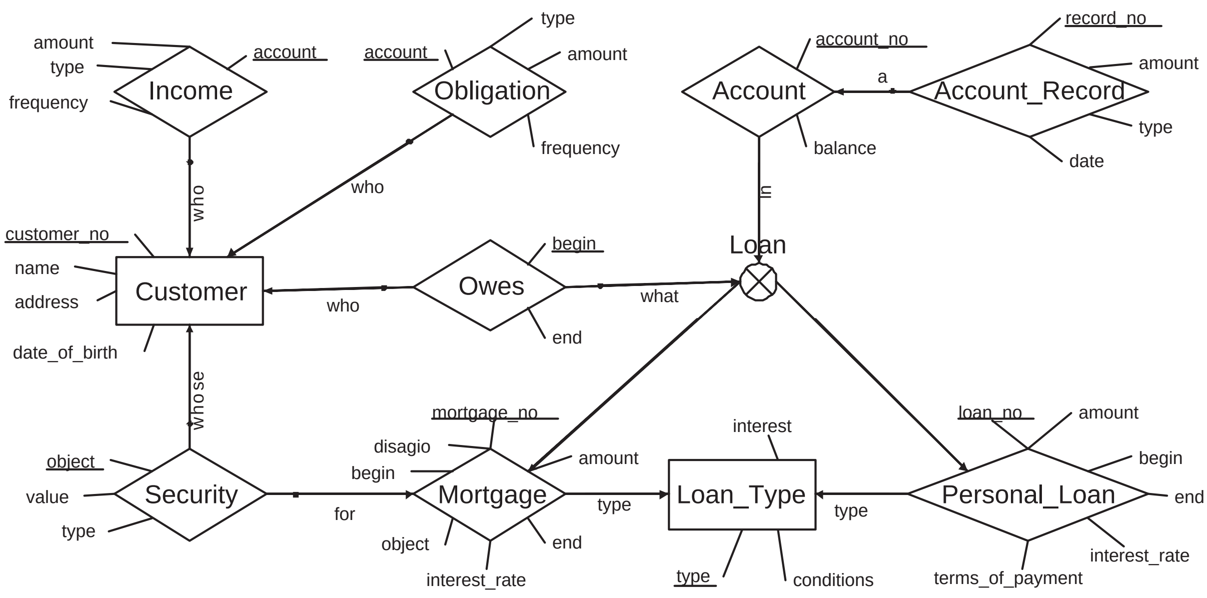

Figure 6.1: HERM diagram for loan application

LOAN_TYPE = (∅, { type, conditions, interest }, { type })

CUSTOMER = (∅, { customer_no, name, address, date_of_birth },
{ customer_no })

PERSONAL_LOAN = ({ type : LOAN_TYPE }, { loan_no, amount,
interest_rate, begin, end, terms_of_payment }, { loan_no })

MORTGAGE = ({ type : LOAN_TYPE }, { mortgage_no, amount, disagio,
interest_rate, begin, end, object }, { mortgage_no })

LOAN = PERSONAL_LOAN ⊕ MORTGAGE

ACCOUNT = ({ ln : LOAN }, { account_no, balance }, { account_no })

ACCOUNT_RECORD = ({ a : ACCOUNT }, { record_no, type, amount,
date }, { a : ACCOUNT, record_no })

OWES = ({ who : CUSTOMER, what : LOAN }, { begin, end },
{ who : CUSTOMER, what : LOAN, begin })

SECURITY = ({ whose : CUSTOMER, for : MORTGAGE }, { value, object,
type }, {whose : CUSTOMER, for : MORTGAGE, object })

INCOME = ({ who : CUSTOMER }, { type, amount, frequency, account },
{ who : CUSTOMER, account })

OBLIGATION = ({ who : CUSTOMER }, { type, amount, frequency,
account }, { who : CUSTOMER, account })

For this it is easy to see that the types CUSTOMER and LOAN_TYPE are on level 0, because they do not have components. Level-1-types are INCOME, OBLIGATION, MORTGAGE and PERSONAL_LOAN, because all their components are on level 0. Consequently, the cluster LOAN ist a level-1-cluster. The types OWES, SECURITY and ACCOUNT are then level-2-types, because all components are on level 1 or below, and finally, ACCOUNT_RECORD is a level-3-type.

Figure 6.1 provides a graphical representation of the schema in Example 6.1. We call this a *HERM diagram*. According to the common convention in Entity-Relationship diagrams we represented types on level 0 by rectangles and types on higher levels by diamonds. Clusters are represented by $\oplus$. We use directed edges from a relationship type to each of its components, and from clusters to their components, and undirected edges between types and their attributes. Roles names are attached to the directed edges. Keys are marked by underlining attributes and marking the edges that correspond to components in the key by filled dots.

As each HERM schema can be represented by such a HERM diagram, i.e. by a graph, we can apply all graph-theoretic notions. In particular, when we talk of *paths* in a HERM schema, we mean a path in the underlying undirected graph that results from ignoring the orientation of edges in the HERM diagram.

In order to define the semantics of HERM schemata we concentrate on identifier-semantics, also known as pointer-semantics. For this assume a countable set ID of identifiers with $ID \cap D = \emptyset$ for all domains D used for simple attributes. Furthermore, we associate with each type $R \in \mathcal{S}$ a *representing attribute*

$$X_R = R(X_{r_1}, \dots, X_{r_n}, A_1, \dots, A_n)$$

with new simple attributes X_{r_i} for each role r_i and $dom(X_{r_i}) = ID$, as well as a *key attribute*

$$K_R = R(X_{r_{i_1}}, \dots, X_{r_{i_\ell}}, A'_1, \dots, A'_m)$$

for key$(R) = \{r_{i_1} : R_{i_1}, \dots, r_{i_\ell} : R_{i_\ell}, A'_1, \dots, A'_m\}$. Obviously, we have $X_R \geq K_R$.

Definition 6.4. An *instance* of a HERM schema $(\mathcal{S}, \Sigma)$ is a family $\{db(R)\}_{R \in \mathcal{S}}$ of finite sets. For each type R the set $db(R)$ consists of pairs (i, v) with $i \in ID$ and $v \in dom(X_R)$ subject to the following conditions:

- Identifiers are locally unique, i.e. whenever $(i, v_1), (i, v_2) \in db(R)$, we must have $v_1 = v_2$.
- Key values are locally unique, i.e. whenever $(i_1, v_1), (i_2, v_2) \in db(R)$ hold with $\pi^{X_R}_{K_R}(v_1) = \pi^{X_R}_{K_R}(v_2)$, then we must have $i_1 = i_2$.
- Roles are always defined, i.e. whenever $(i, v) \in db(R)$ and $\pi^{X_R}_{R(X_{r_j})}(v) = i'$ for $r_j : R_j \in$ comp(R), then $(i', v') \in db(R_j)$ for some $v' \in dom(X_{R_j})$.
- The integrity constraints in Σ are satisfied.

For each cluster $C = R_1 \oplus \cdots \oplus R_k$ the set $db(C)$ is the disjoint union of the sets $db(R_i)$ $(i = 1, \dots, k)$.

We write $inst(\mathcal{S}, \Sigma)$ for the *set of all instances* over $(\mathcal{S}, \Sigma)$.

6.2 Query Languages for HERM

As for the relational data model, basic queries against a HERM schema can be formulated both in an algebraic and a logical way. We will extend both the simple HERM algebra and the HERM calculus in a way that we can express more queries, but let us start with first-order queries.

Definition 6.5. The *HERM algebra* $\mathcal{H}$ provides the operations σ_φ (selection) with a selection formula φ, $\pi_{A_1,\dots,A_m}$ (projection) with a generalised subset $\{A_1,\dots,A_m\}$, ϱ_f (renaming) with a renaming function f, $\bowtie_G$ (join) with a common generalised subset G, $\cup$ (union), $-$ (difference), $\nu_{X:A_1,\dots,A_n}$ with attributes $A_1,\dots,A_n$ (nest), and μ_A (unnest) with a set attribute A.

As the details of these operations are not much different from the relational data model, we omit the details and refer to [106].

However, in order to make the HERM algebra operational for our purposes here, we need a little extension:

- We permit new type names R to be added to $\mathcal{S}$, and use assignments $R := exp$ with a HERM algebra expression exp. Then applying such a query to an instance of $(\mathcal{S},\Sigma)$ results in an instance over $(\mathcal{S}\cup\{R\},\Sigma)$. The type or cluster definition for R is implicitly determined by the expression exp.
- In order to satisfy the uniqueness of identifiers, such assignments involve the non-deterministic creation of identifiers in *ID* for the pairs (i,v) in the added $db(R)$. Such identifier creation has been investigated thoroughly in [115].
- While sequences of assignments only extend the schema, we also allow dropping types or clusters.

In summary, a *HERM algebra program* P has the form $C_1;\dots;C_r \setminus \mathcal{S}'$, where each C_i is an assignment, say $R_i := exp_i$ that extends the schema by R_i and the instance by $db(R_i)$, while $\mathcal{S}'$ is a subschema of $\mathcal{S}\cup\{R_1,\dots,R_r\}$. Thus, P defines a mapping $q(P)$ taking instances over $(\mathcal{S},\Sigma)$ to instances over $(\mathcal{S}',\Sigma')$, though the set Σ' of constraints is left implicit.

The algebra may be further extended with `WHILE` (which introduces a fix point operator), in which case we talk of the *extended HERM algebra* $\mathcal{H}_{ext}$. In this case we add constructs of the form

`WHILE` change `DO` $C_1;\dots;C_r$ `END`.

We obtain a logical perspective from the following simple observation. Each type $R\in\mathcal{S}$ – or more precisely, its representing attribute X_R – defines a variable in a higher-order logic. The order depends on the depth of nesting of the attributes in attr(R). For instance, if the set constructor is not used, we obtain a first-order variable. In fact, we obtain a *type* for each such variable, where types correspond to nested attributes. Thus, a schema defines a signature, and each instance defines a finite structure for this signature.

There are a few subtleties to be aware of. First, for clusters we have to allow particular union variables to cope with the requirement to have disjoint unions. Second, we have to define first the logic and then permit only those structures that are in fact models for the theory defined by the restrictions in Definition 6.4.

Now use further variables and constants of any available type and define atoms as follows:

- A *predicative atom* has the form $X(t_1,\dots,t_n)$, which evaluates to true if the tuple $(t_1,\dots,t_n)$ lies in X, with a higher-order variable X and *terms*, i.e. variables or constants, $t_1,\dots,t_n$ such that the types of the t_i and the one of X match properly.
- An *equational atom* has the form $t_1 = t_2$ with terms of the same type or of different types, where one is a cluster and the other one is one of its components.

Finally, use the usual connectives $\wedge$, $\vee$, $\rightarrow$, $\exists$ and $\forall$ to define *HERM logic*. Then the concept of free and bound variables is defined as usual. We write $fr(\varphi)$ for the set of free variables of a formula φ.

Definition 6.6. A *HERM calculus query* has the form $X(x_0, \dots, x_n) : \varphi$ with a formula φ of HERM logic such that $fr(\varphi) \subseteq \mathcal{S} \cup \{x_1, \dots, x_n\}$ and $\{x_1, \dots, x_n\} \subseteq fr(\varphi)$.

Obviously, we may interpret a formula φ provided we are given a value assignment $\sigma(x_i)$ for all the variables $x_1, \dots, x_n$ and an instance over $(\mathcal{S}, \Sigma)$. If according to this interpretation the formula φ is interpreted as true, we obtain a tuple $(\sigma(x_0), \dots, \sigma(x_n))$ with a new identifier $\sigma(x_0) \in ID$. The result is the set of all such tuples, and will be bound to variable X.

It has been shown in [106] that such HERM calculus queries express exactly the same as HERM algebra queries with non-deterministic identifier creation and assignments to new type variables. Using sequences and a fixed-point construction yields the same expressiveness as the extended HERM algebra. We use the term *extended HERM calculus* for this approach to query languages.

6.3 Schema Dominance and Equivalence

Now assume we have two schemata $(\mathcal{S}_1, \Sigma_1)$ and $(\mathcal{S}_2, \Sigma_2)$ that are to be integrated. By "polishing" the schemata we may assume that there are no name conflicts, so we can build the disjoint union $\mathcal{S}_1 \oplus \mathcal{S}_2$. Any associations between the two schemata can be expressed using another set of constraints Σ_0, i.e. we look in fact at $(\mathcal{S}_1 \oplus \mathcal{S}_2, \Sigma_1 \cup \Sigma_2 \cup \Sigma_0)$, which becomes subject to a restructuring. Basically, we are aiming at removing redundancy between the two schemata (expressed via Σ_0), i.e. we want to replace the union of the schemata by a new schema $(\mathcal{S}, \Sigma)$ that is equivalent to the given one. Thus, we first need a notion of schema equivalence.

Definition 6.7. A schema $(\mathcal{S}', \Sigma')$ *dominates* another schema $(\mathcal{S}, \Sigma)$ by means of the language $\mathcal{L}$ (notation: $(\mathcal{S}, \Sigma) \sqsubseteq_{\mathcal{L}} (\mathcal{S}', \Sigma')$) iff there are mappings $f : \mathit{inst}(\mathcal{S}, \Sigma) \rightarrow \mathit{inst}(\mathcal{S}', \Sigma')$ and $g : \mathit{inst}(\mathcal{S}', \Sigma') \rightarrow \mathit{inst}(\mathcal{S}, \Sigma)$ both expressed in $\mathcal{L}$ such that the composition $g \circ f$ is the identity.

Definition 6.8. If we have $(\mathcal{S}, \Sigma) \sqsubseteq_{\mathcal{L}} (\mathcal{S}', \Sigma')$ as well as $(\mathcal{S}', \Sigma') \sqsubseteq_{\mathcal{L}} (\mathcal{S}, \Sigma)$, we say that the two schemata are *equivalent* with respect to $\mathcal{L}$ (notation: $(\mathcal{S}, \Sigma) \cong_{\mathcal{L}} (\mathcal{S}', \Sigma')$).

According to [75] there are various choices for the language $\mathcal{L}$ leading to suitable notions of dominance and equivalence. Here, we only consider the most general one, in which case f and g would be expressed by computable functions. Note that this notion of dominance is exactly what we employed for schema refinement in TASMs.

Lemma 6.9. *Let $\mathfrak{M}$ be schema refined to $\mathfrak{M}'$ using some abstraction predicate $\mathcal{A}$, and let $\mathcal{S}, \mathcal{S}'$ be the schemas of $\mathfrak{M}, \mathfrak{M}'$. Then $\mathcal{S}'$ dominates $\mathcal{S}$.*

Proof. By definition of schema refinement there exist computable functions f, g

$$f \subseteq \mathcal{A} \subseteq g^{-1}$$

It follows immediately that $g \circ f$ is the identity mapping, which shows dominance. □

Let us adopt this definition to view integration. First recall that a view is nothing but a stored query. More precisely, a *view* V on a schema $(\mathcal{S}, \Sigma)$ consists of a schema $\mathcal{S}_V$ and a query q_V with a query mapping $inst(\mathcal{S}, \Sigma) \rightarrow inst(\mathcal{S}_V)$. Here, $inst(\mathcal{S}, \Sigma)$ denotes the set of instances of schema $(\mathcal{S}, \Sigma)$.

So the view integration problem starts with two views V_1 and V_2 on the same schema $(\mathcal{S}, \Sigma)$, and should result in a new integrated view V such that $\mathcal{S}_V$ results from integration of the schemata $\mathcal{S}_{V_1}$ and $\mathcal{S}_{V_2}$, and for each instance db over $(\mathcal{S}, \Sigma)$ the two query results $q_{V_1}(db)$ and $q_{V_2}(db)$ together are equivalent to $q_V(db)$.

Now, if the schemata $\mathcal{S}_{V_1}$ and $\mathcal{S}_{V_2}$ are "cleaned", we may combine the queries q_{V_1} and q_{V_2} into one yielding a query mapping $inst(\mathcal{S}, \Sigma) \rightarrow inst(\mathcal{S}_{V_1} \cup \mathcal{S}_{V_2})$ defined by the query $q_{V_1} \cup q_{V_2}$. If we simply integrate the schemata $\mathcal{S}_{V_1}$ and $\mathcal{S}_{V_2}$ into $\mathcal{S}_V$ according to the method described above, we obtain a computable function $f : inst(\mathcal{S}_{V_1} \cup \mathcal{S}_{V_2}) \rightarrow inst(\mathcal{S}_V)$, which constitutes a query mapping $q_f = f$. Taking $q_V = q_f \circ (q_{V_1} \cup q_{V_2})$, V becomes a view over $(\mathcal{S}, \Sigma)$ with schema $\mathcal{S}_V$ and defining query q_V. Since q_f, q_{V_1}, q_{V_2} are all computable and have computable inverses, this also holds true for q_V. Thus $\mathcal{S}_V$ dominates $\mathcal{S}_{V_1}, \mathcal{S}_{V_2}$.

This approach to view integration also works in the more general situation, where the given views V_1 and V_2 are defined over different schemata $(\mathcal{S}_1, \Sigma_2)$ and $(\mathcal{S}_2, \Sigma_2)$, respectively.

6.4 Schema and View Integration Process

We first describe a pragmatic method for guiding the schema integration process with explanations on how this method applies to view integration. Then the transformation rules and the formal definition of the rules are described in the following section.

1. The first step is the homogenisation of the schemata. This includes the restructuring of the schemata turning attributes into entity types, entity types into relationship types and vice versa. Furthermore, we add attributes and shift attributes along hierarchies and paths. All these individual paces correspond to the application of transformation rules. The result of the homogenisation step are schemata $(\mathcal{S}'_1, \Sigma'_1)$ and $(\mathcal{S}'_2, \Sigma'_2)$.

2. The second step consists in adding inter-schema integrity constraints that describe the semantic relationships between the two schemata. Formally, we obtain another set of constraints Σ_0, and thus the result of this step is a single HERM schema $(\mathcal{S}'_1 \cup \mathcal{S}'_2, \Sigma'_1 \cup \Sigma'_2 \cup \Sigma_0)$.

3. The third step is only a preparation of the following steps. Due to the expected large size of the schemata, these are divided into modules, each of which describes a sub-schema. Corresponding modules are identified in order to approach the integration of modules first. If schemata are of moderate size, this step can be omitted.

4. Step four considers the integration of types on level 0, 1, etc., i.e. we start with entity types and level-0-clusters, then proceed with relationship types and clusters on level 1, then relationship types and clusters on level 2, etc. For each level we integrate corresponding types or clusters with respect to equality, containment, overlap and disjointness conditions. Note that this step is similar to the work done in [60, 63, 104].

5. The fifth step deals with the integration of paths using path inclusion dependencies.

6. Finally, we consider remaining integrity constraints such as (path) functional dependencies and join dependencies.

6.5 Transformation Rules

The set of rules in this section covers the rules that have been stated implicitly or explicitly in former work by others. In particular, they are closest to the set of rules we presented in [75], with small modifications as follows: in ***Rule*** 7, a constraint is added to ensure no part of the key is removed; in ***Rule*** 9, the constraint is extended to ensure consistency between two referencing relationships; ***Rule*** 13 for adding a new type is included; ***Rule*** 28 is simplified by restricting it to a single multi-valued dependency (for multiple dependencies we can simply apply ***Rule*** 28 multiple times); and some typos are corrected in ***Rule*** 19, 20, 22, 23, 24.

As there are many ways to integrate schemas for preserving equivalence and achieving dominance, the set of transformation rules we provide is not complete. Our goal was to obtain a set of rules which cover most cases occurring in practice. The set of rules can be extended of cause, if such a need arises. We also did not attempt to minimize it - even if a rule is technically redundant, it can still make the transformation process easier.

Our only provable claim (we also claim that they cover most practical cases, but that's impossible to show formally) is that the set of rules are correct by the notion of schema equivalence and dominance. This is established in the formalisation of the rules in TASM by constructing the abstraction predicate $\mathcal{A}$ and corresponding computable functions f and g, which show that they produce a (strong) schema refinement. In principal, we would need to show that f, g are computable, that $f \subseteq \mathcal{A} \subseteq g^{-1}$ holds, and for strong schema refinement that $\mathcal{A}$ is invariant under isomorphisms. However, for all rules given these properties will be fairly obvious, thus we omit such proofs.

The set of rules that we provided are designed for schema transformations. Hence, data changes are only modelled in the rules. However, the corresponding changes to the operations are incorporated in the refinement-based development process and the rules for dialogue objects.

Furthermore, the set of rules are derived from a pragmatic process, i.e. the five-step process as described in 6.4, which can be used as a general guidelines for application of the rules.

In the following we will describe the transformation rules in detail. In particular, we assume a given HERM schema $(\mathcal{S}, \Sigma)$, but each rule models the involved schemas only, with the resulting schema being $(\mathcal{S}_{new}, \Sigma_{new})$. The new types in the new schema will be marked with a subscript $_{new}$. For better explanations we use the example Figures 6.1 from [75] in some of the rules.

We state these rules in TASM in the form

$$\frac{\mathfrak{M} \rhd aFunc = \ldots,\ \ldots}{\mathfrak{M}^* \rhd newFunc = \ldots,\ \ldots}\ \varphi$$

with lists of function declarations above and below the bar. These lists may also include type definitions, which we use to model clusters. The symbol $=_t$ is used to distinguish type definitions from declarations of basic function.

The meaning of such a rule is that under some side conditions φ, parts of the machine $\mathfrak{M}$ will be replaced by new functions in a refining machine $\mathfrak{M}^*$. That is, all functions above the bar will be removed, while all functions below the bar will be added to to the schema of $\mathfrak{M}^*$. The use of placeholders for function names, labels and types, which can then be instantiated for concrete refinements, is common practice, thus we omit a more formal definition of our language for refinement rules.

We restrict the refinements to be on the set of functions that represent the relations in the machines, i.e. are part of the schema. In the refinement rules dealing with keys, we assume keys to be defined as supertypes of relations.

We then proceed with specifying the side condition under which the rule can be applied, the constraints which the new machine must comply with, and the corresponding abstraction predicate $\mathcal{A}$ and the two computable functions f and g for (strong) schema refinement.

6.5.1 Schema Restructuring

The first group of rules addresses the aspect of schema restructuring which will be used in the homogenisation step 1 of our method.

Rule 1. Replace a tuple attribute $X(A_1, \dots, A_m)$ in an entity or relationship type R by the attributes $A_1, \dots, A_m$. The resulting type R_{new} will replace R. For $X(A'_1, \dots, A'_n) \in key(R)$ with $A_i \leq A'_i$ we obtain $A'_1, \dots, A'_n \in key(R_{new})$.

$$\frac{\mathfrak{M} \rhd R = X(t_{11} \times \cdots \times t_{1m}) \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\}}{\mathfrak{M}^* \rhd \quad R_{new} = t_{11} \times \cdots \times t_{1m} \times t_2 \times \cdots \times t_n \to \{\mathbb{1}\}}$$

with the constraints:

$$A'_1, \dots, A'_l \in key(R_{new}) \text{ if } X(A'_1, \dots, A'_l) \in key(R)$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x_1, \dots, x_{n+m-1}.(R_{new}(x_1, \dots, x_{n+m-1}) = 1$$
$$\Leftrightarrow R(X(x_1, \dots, x_m), \dots, x_{n+m-1}) = 1)$$

The corresponding computable queries f and g are constructed as

$$R_{new} := \{(x_1, \dots, x_{n+m-1}) \mid (X(x_1, \dots, x_m), \dots, x_{n+m-1}) \in R\}$$

and

$$R := (\{X(x_1, \dots, x_m), \dots, x_{n+m-1}) \mid (x_1, \dots, x_{n+m-1}) \in R_{new}\},$$

respectively.

There is no side condition for the rule to be applied.

Example 6.2. *Assume an entity type* Customer *with a tuple attribute* name *as follows:*

Customer = (∅, { customer_no, name(first-name,last-name), address, date_of_birth }, { customer_no })

Applying Rule 1, we have

Customer = (∅, { customer_no, first-name,last-name, address, date_of_birth }, { customer_no })

□

This rule includes the simple case, where R is an entity type, which could be treated as a separate rule.

Rule 2. Replace a component $r : R'$ in a relationship type R by lower level components and attributes. Let the new type be R_{new}. For $\text{comp}(R') = \{r_1 : R_1, \dots, r_n : R_n\}$ we get $\text{comp}(R_{new}) = \text{comp}(R) - \{r : R'\} \cup \{r_1^{(r)} : R_1, \dots, r_n^{(r)} : R_n\}$ with new role names $r_i^{(r)}$ composed from r_i and r and $\text{attr}(R_{new}) = \text{attr}(R) \cup \text{attr}(R')$. In the case $r : R' \in \text{key}(R)$ and $\text{key}(R') = \{r_{i_1} : R_{i_1}, \dots, r_{i_k} : R_{i_k}, A_1, \dots, A_m\}$ we obtain $\text{key}(R_{new}) = \text{key}(R) - \{r : R'\} \cup \{r_{i_1}^{(r)} : R_{i_1}, \dots, r_{i_k}^{(r)} : R_{i_k}, A_1, \dots, A_m\}$, otherwise we have $\text{key}(R_{new}) = \text{key}(R)$.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R' = r' : id \times t' \to \{\mathbb{1}\} \\ & R = r' : ref \times t \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & R' = r' : id \times t' \to \{\mathbb{1}\} \\ & R_{new} = t' \times t \to \{\mathbb{1}\}\end{array}}$$

with the constraints:

$$\text{key}(R_{new}) = \begin{cases} \text{key}(R) - \{r : R'\} \cup \text{key}(R') & \text{if } r' \in \text{key}(R) \\ \text{key}(R) & \text{otherwise} \end{cases}$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x, x'.(R_{new}(x', x) = 1 \Leftrightarrow \exists r.(R'(r, x') = 1 \wedge R(r, x) = 1))$$

The corresponding computable queries f and g are constructed as

$$R_{new} := \{(x, x') \mid \exists r.((r, x) \in R \wedge (r, x') \in R')\}$$

and

$$R := \{(r, x) \mid \exists x'.((x', x) \in R_{new} \wedge (r, x') \in R')\},$$

respectively.

There is no side condition for the rule to be applied.

Example 6.3. *Let us look at the relationship types* MORTGAGE *and* SECURITY *and the entity type* LOAN_TYPE *in Figures 6.1:*

LOAN_TYPE = (∅, { type, conditions, interest }, { type })
MORTGAGE = ({ type : LOAN_TYPE }, { mortgage_no, amount, disagio,
interest_rate, begin, end, object }, { mortgage_no })
SECURITY = ({ whose : CUSTOMER, for : MORTGAGE }, { value, object,

type }, {whose : Customer, for : Mortgage, object })

Applying Rule 2 on Security *to replace the component* for : Mortgage*, we obtain*

Security = ({ whose : Customer, for_type : Loan_Type }, { value, object, type, mortgage_no, amount, disagio, interest_rate, begin, end, object }, {whose : Customer, for_type : Loan_Type, object })

$\square$

Rule 3. Replace a component $r : R'$ in a relationship type R by lower level components and attributes. Let the new type be R_{new}. For R' is an entity type, we get $\text{comp}(R_{new}) = \text{comp}(R) - \{r : R'\}$ with $\text{attr}(R_{new}) = \text{attr}(R) \cup \text{attr}(R')$. In the case $r : R' \in \text{key}(R)$ and $\text{key}(R') = \{A_1, \ldots, A_m\}$ we obtain $\text{key}(R_{new}) = \text{key}(R) - \{r : R'\} \cup \{A_1, \ldots, A_m\}$, otherwise we have $\text{key}(R_{new}) = \text{key}(R)$.

This is a simplified rule of Rule 2 in the case, where R' is an entity type. We omit the details here.

Rule 4. Replace a cluster $C = C_1 \oplus \cdots \oplus C_n$ with a cluster component $C_i = C_{i_1} \oplus \cdots \oplus C_{i_m}$ by a new cluster $C = C_1 \oplus \cdots \oplus C_{i-1} \oplus C_{i_1} \oplus \cdots \oplus C_{i_m} \oplus C_{i+1} \oplus \cdots \oplus C_n$.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & C =_t C_1 \oplus \ldots C_i \oplus \cdots \oplus C_n \\ & C_i =_t C_{i_1} \oplus \cdots \oplus C_{i_m}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & C_i =_t C_{i_1} \oplus \cdots \oplus C_{i_m} \\ & C =_t C_1 \oplus \cdots \oplus C_{i-1} \oplus C_{i_1} \oplus \cdots \oplus C_{i_m} \oplus C_{i+1} \oplus \cdots \oplus C_n\end{array}}$$

Since a cluster is not a relation but a type definition, this rule does not change the structure of the relation. Thus there is no side condition, or corresponding abstract predicate, or computable functions required.

Example 6.4. *Let us assume the following:*

Account = personal ⊕ Business
personal = deposit ⊕ cheque.

Applying Rule 4 on Account*, we obtain*

Account = deposit ⊕ cheque ⊕ Business.

$\square$

Rule 5. Replace a relationship type R with a cluster component $r : C$ ($C = C_1 \oplus \cdots \oplus C_n$) by a new cluster $C_{new} = R_{1,new} \oplus \cdots \oplus R_{n,new}$ and new relationship types $R_{i,new}$ with $comp(R_{i,new}) = comp(R) - \{r : C\} \cup \{r_i : C_i\}$ and $attr(R_{i,new}) = attr(R)$. For $r : C \in key(R)$ we obtain $key = key(R) - \{r : C\} \cup \{r : C_i\}$, otherwise take $key = key(R)$.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R = C \times t \to \{\mathbb{1}\} \\ & C =_t C_1 \oplus \cdots \oplus C_n\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & C =_t C_1 \oplus \cdots \oplus C_n \\ & C_{new} =_t r_{1,new} : ref \oplus \cdots \oplus r_{n,new} : ref \\ & R_{1,new} = r_{1,new} : id \times C_1 \times t \to \{\mathbb{1}\} \\ & \ldots \\ & R_{n,new} = r_{n,new} : id \times C_n \times t \to \{\mathbb{1}\}\end{array}}$$

with the constraints:

$$\text{key}(R_{i,new}) = \begin{cases}\text{key}(R) - \{r : C\} \cup \{r_i : C_i\} & \text{if } r \in \text{key}(R) \\ \text{key}(R) & \text{otherwise}\end{cases}$$

for all i such that $1 \leq i \leq n$.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x, r.(\exists id.R_{i,new}(id, r, x) = 1 \Leftrightarrow R((1, r), x) = 1 \wedge \\ \cdots \wedge \exists id.R_{n,new}(id, r, x) = 1 \Leftrightarrow R((n, r), x) = 1)$$

The corresponding computable queries f and g:

$$R_{1,new} := \{(id, r, x) \mid ((1, r), x) \in R\} \\ \ldots \\ R_{n,new} := \{(id, r, x) \mid ((n, r), x) \in R\}$$

and

$$R := \bigcup_{i=1}^{n} \{(r, x) \mid \exists id.((id, r, x) \in R_{i,new})\}$$

respectively.

There is no side condition for the rule to be applied. However, we need to change all relations referencing R to reference C_{new} instead.

Example 6.5. *In the loan application of Section 6.1 we have:*

ACCOUNT = ({ ln : LOAN }, { account_no, balance }, { account_no })

LOAN = HOME_LOAN ⊕ MORTGAGE

PERSONAL_LOAN = ({ type : LOAN_TYPE }, { loan_no, amount, interest_rate, begin, end, terms_of_payment }, { loan_no })

MORTGAGE = ({ type : LOAN_TYPE }, { mortgage_no, amount, disagio, interest_rate, begin, end, object }, { mortgage_no })

Applying the Rule 5 on the relationship type Account*, we get:*

Account = Personal_Account ⊕ Mortgage_Account
Personal_Account = ({ for : Personal_Loan },
{ account_no, balance }, { account_no })
Mortgage_Account = ({ for : Mortgage }, { account_no, balance },
{ account_no })

□

In the case of the restructuring Rules 1 – 5 we get the resulting schema equivalent to the original schema. The next rule only guarantees that the resulting schema dominates the old one.

Rule 6*.* Replace a key-based inclusion dependency $R'[key(R')] \subseteq R[key(R)]$ by new relationship types R_{new} with $comp(R_{new}) = \{r' : R', r : R\} = key(R_{new})$ and $attr(R_{new}) = \emptyset$ together with participation cardinality constraints $card(R_{new}, R) = (0,1)$ and $card(R_{new}, R') = (1,1)$.

$$\frac{\begin{array}{rl} \mathfrak{M} \rhd & R = r : id \times t \to \{\mathbb{1}\} \\ & R' = r' : id \times t' \to \{\mathbb{1}\} \end{array}}{\begin{array}{rl} \mathfrak{M}^* \rhd & R = r : id \times t \to \{\mathbb{1}\} \\ & R' = r' : id \times t' \to \{\mathbb{1}\} \\ & R_{new} = r' : ref \times r : ref \to \{\mathbb{1}\} \end{array}} \varphi$$

with the side condition φ:

$$R'[key(R')] \subseteq R[key(R)]$$

The constraints:

$$card(R'_{new}, R) = (0,1) \wedge card(R'_{new}, R') = (1,1)$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall r, r'.(R_{new}(r', r) = 1 \Leftrightarrow \exists x, x'.(R(r,x) = 1 \wedge R'(r', x') = 1)$$

The corresponding computable queries f:

$$R_{new} := \{(r', r) \mid \exists x, x'.(r,x) \in R \wedge (r', x') \in R'\}$$

and g is an identity function since nothing has changed in relation R and R' in the refined machine.

Example 6.6*. Let us assume two relationship types* Staff *and* Supervisor *as follows:*

Staff = (∅, { staff_id, start_date, dept_id }, { staff_id })
Supervisor = (∅, { staff_id, student_id, start_date, end_date }, { staff_id })

To apply Rule 6 on the following dependency:

$\textsc{Supervisor}[key(\textsc{Supervisor})] \subseteq \textsc{Staff}[key(\textsc{Staff})]$

We obtain a new relationship type STAFF_SUPERVISOR *and two cardinality constraints as follows:*

$\textsc{Staff\_supervisor} = (\,\{\,\text{as:}\textsc{Supervisor},\ \text{being:}\textsc{Staff}\,\}, \emptyset, \{\,\text{as:}\textsc{Supervisor},\ \text{being:}\textsc{Staff}\,\}\,)$

$card(\textsc{Staff\_supervisor}, \textsc{Staff}) = (0, 1)$

$card(\textsc{Staff\_supervisor}, \textsc{Supervisor}) = (1, 1)$

□

The last two restructuring rules allow to switch between attributes and entity types and between entity and relationship types. The Rule 7 and 8 guarantee schema equivalence.

Rule 7. Replace an entity type E with $A \in attr(E)$ and $A \notin key(E)$ by E_{new} such that $attr(E_{new}) = attr(E) - \{A\}$ and $key(E_{new}) = key(E)$ hold. Furthermore, introduce an entity type E'_{new} with $attr(E'_{new}) = \{A\} = key(E'_{new})$ and a new relationship type R_{new} with $comp(R_{new}) = \{r_{new} : E_{new}, r'_{new} : E'_{new}\} = key(R_{new})$ and $attr(R_{new}) = \emptyset$. Add the cardinality constraints $card(R_{new}, E_{new}) = (1, 1)$ and $card(R_{new}, E'_{new}) = (1, \infty)$.

$$\frac{\mathfrak{M} \rhd E = A : t_a \times t \to \{\mathbb{1}\}}{\begin{array}{ll}\mathfrak{M}^* \rhd & E_{new} = r_1 : id \times A : t_a \to \{\mathbb{1}\} \\ & E'_{new} = r_2 : id \times t \to \{\mathbb{1}\} \\ & R_{new} = r_1 : ref \times r_2 : ref \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$A \notin key(E)$$

The constraints:

$$key(E_{new}) = key(E) \wedge card(R_{new}, E_{new}) = (1, 1) \wedge card(R_{new}, E'_{new}) = (1, \infty)$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall a, x\ (\exists r_1, r_2.(E_{new}(r_1, a) = 1 \wedge E'_{new}(r_2, x) = 1 \wedge R_{new}(r_1, r_2) = 1) \Leftrightarrow E(a, x) = 1)$$

The corresponding computable queries f and g:

$$\begin{aligned} E_{new} &:= \{(h(a), a) \mid \exists r, x.(r, a, x) \in E\} \| \\ E'_{new} &:= \{(r, x) \mid \exists a.(r, a, x) \in E\} \| \\ R_{new} &:= \{(h(a), r) \mid \exists x.(r, a, x) \in E\} \end{aligned}$$

and

$$E := \{(a, x) \mid \exists r, r'.((r, a) \in E_{new} \wedge (r', x) \in E'_{new} \wedge (r, r') \in R_{new})\},$$

respectively, for some fixed computable injective function $h : A : t_a \to r_1 : id$.

Example 6.7. *Let us look at the entity type:*

CUSTOMER = (∅, { customer_no, name, address, date_of_birth },
{ customer_no })

we then take address *as the attribute A, applying the Rule 7, we obtain:*

CUSTOMER_NEW = (∅, { customer_no, name, date_of_birth },
{ customer_no })

CUSTOMER′_NEW = (∅, { address }, { address })

C_ADDRESS = ({ c:CUSTOMER_NEW, a:CUSTOMER′_NEW },
∅, { c:CUSTOMER_NEW, a:CUSTOMER′_NEW })

with the cardinality constraints:

*card(*C_ADDRESS,CUSTOMER_NEW*) = (1,1) and*
*card(*C_ADDRESS,CUSTOMER′_NEW*)* = $(1, \infty)$

□

Rule 8. Replace a relationship type R with $comp(R) = \{r_1 : R_1, \ldots, r_n : R_n\}$ and the cardinality constraints $card(R, R_i) = (x_i, y_i)$ by a new entity type E_{new} with $attr(E_{new}) = attr(R) = key(E_{new})$ and n new relationship types $R_{i,new}$ with $comp(R_{i,new}) = \{r_i : R_i, r : E_{new}\} = key(R_{i,new})$ and $attr(R_{i,new}) = \emptyset$. Replace the cardinality constraints by $card(R_{i,new}, R_i) = (1, y_i)$ and $card(R_{i,new}, E_{new}) = (1, \infty)$.

$$\frac{\mathfrak{M} \rhd R = r : id \times r_1 : ref \times \cdots \times r_n : ref \times t \to \{\mathbb{1}\}}{\begin{array}{ll}\mathfrak{M}^* \rhd & E_{new} = e : id \times t \to \{\mathbb{1}\} \\ & R_{1,new} = r_1 : ref \times e : ref \to \{\mathbb{1}\} \\ & \ldots \\ & R_{n,new} = r_n : ref \times e : ref \to \{\mathbb{1}\}\end{array}} \varphi$$

with the the side condition φ:

$$comp(R) \cap key(R) = \emptyset$$

and the constraints:

- $key(E_{new}) = key(R), key(R_{i,new}) = \{r : E_{new}\}$;
- $card(R_{i,new}, R_i) = (x_i, y_i)$ and $card(R_{i,new}, E_{new}) = (1, 1)$, where $card(R, R_i) = (x_i, y_i)$

The corresponding abstraction predicate $\mathcal{A}$:
$\forall r_1, \dots, r_n, x.$
$(\exists e.(E_{new}(e, x) = 1 \wedge R_1(r_1, e) = 1 \wedge \dots R_n(r_n, e) = 1) \Leftrightarrow R(r_1, \dots, r_n, x) = 1)$

The corresponding computable queries f and g:

$$\begin{aligned} E_{new} &:= \{(e, x) \mid \exists r_1, \dots, r_n.(e, r_1, \dots, r_n, x) \in R\} \| \\ R_{1,new} &:= \{(r_1, e) \mid \exists r_2, \dots, r_n, x.(e, r_1, \dots, r_n, x) \in R\} \| \dots \| \\ R_{n,new} &:= \{(r_n, e) \mid \exists r_1, \dots, r_{n-1}, x.(e, r_1, \dots, r_n, x) \in R\} \| \end{aligned}$$

and

$$R := \{(e, r_1, \dots, r_n, x) \mid (e, x) \in E_{new} \wedge (r_1, e) \in R_{1,new} \wedge \dots \wedge (r_n, e) \in R_{n,new}\},$$

respectively.

Example 6.8. *Let us look at the loan application of Section 6.1 again where we have the relationship type and the cardinality constraints as follows:*

OWES = ({ who : CUSTOMER, what : LOAN }, { begin, end },
{ who : CUSTOMER, what : LOAN, begin })

card(OWES,CUSTOMER) = (0,m)
card(OWES,LOAN) = (0,n)

Applying Rule 8, we obtain:

PERIOD = (∅, { begin, end }, {begin, end })
CUSTOMER_NEW = ({ who: CUSTOMER, when: PERIOD }, ∅,
{ who: CUSTOMER, when: PERIOD })
LOAN_NEW = ({ what: LOAN, when: PERIOD }, ∅, { what: LOAN,
when: PERIOD })

and cardinality constraints:

$card$(CUSTOMER_NEW, CUSTOMER) $= (1, m)$
$card$(CUSTOMER_NEW, PERIOD) $= (1, \infty)$
$card$(CUSTOMER_NEW, LOAN) $= (1, n)$
$card$(LOAN_NEW, PERIOD) $= (1, \infty)$

□

In the case of Rule 8 explicit knowledge of the key of R allows to sharpen the cardinality constraints.

6.5.2 Shifting Attributes

The second group of rules deals with the shifting of attributes. This will also be used in the homogenisation step 1 of our method. Rule 9 allows to shift a synonymous attribute occurring in two subtypes, i.e. whenever tuples agree on the key they also agree on that attribute, to be shifted to a supertype. This rule leads to a dominating schema. Conversely, Rule 10 allows to shift an attribute from a supertype to subtypes, in which case schema equivalence can be verified.

***Rule** 9.* For $comp(R_i) = \{r_i : R\}$ and $A \in attr(R_i) - key(R_i)$ $(i = 1, 2)$ together with the constraint $\forall t, t' \in R_1 \cup R_2.t[r] = t'[r] \Rightarrow t[A] = t'[A] \wedge \forall t \in R.\ \exists t' \in R_1 \cup R_2.t'[r] = t$ replace the types R, R_1 and R_2 such that $attr(R_{new}) = attr(R) \cup \{A\}$, $comp(R_{i,new}) = \{r_i : R_{new}\}$ and $attr(R_{i,new}) = attr(R_i) - \{A\}$ hold.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R = r : id \times t \to \{\mathbb{1}\} \\ & R_1 = A : t_a \times t_1 \times r : ref \to \{\mathbb{1}\} \\ & R_2 = A : t_a \times t_2 \times r : ref \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & R_{new} = r : id \times A : t_a \times t \to \{\mathbb{1}\} \\ & R_{1,new} = t_1 \times r : ref \to \{\mathbb{1}\} \\ & R_{2,new} = t_2 \times r : ref \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$\forall t, t' \in R_1 \cup R_2.t[r] = t'[r] \Rightarrow t[A] = t'[A] \ \wedge \forall t \in R.\ \exists t' \in R_1 \cup R_2.t'[r] = t[r]$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\begin{gathered}\forall a, x_1, x_2, x.((\exists r.R_{new}(r, a, x) = 1 \wedge R_{1,new}(x_1, r) = 1 \\ \Leftrightarrow \exists r.(R(r, x) = 1 \wedge R_1(a, x_1, r) = 1) \wedge \\ (\exists r.R_{new}(r, a, x) = 1 \wedge R_{2,new}(x_2, r) = 1 \\ \Leftrightarrow \exists r.(R(r, x) = 1 \wedge R_2(a, x_2, r) = 1))\end{gathered}$$

The corresponding computable queries f and g:

$$\begin{gathered}R_{1,new} := \{(x_1, r) \mid \exists a.(a, x_1, r) \in R_1\}\| \\ R_{2,new} := \{(x_2, r) \mid \exists a.(a, x_2, r) \in R_2\}\| \\ R_{new} := \{(r, a, x) \mid (r, x) \in R \wedge (\exists x_1.(a, x_1, r) \in R_1 \vee \exists x_2.(a, x_2, r) \in R_2))\}\end{gathered}$$

and

$$\begin{gathered}R_1 := \{(a, x_1, r) \mid \exists x.(r, a, x) \in R_{new} \wedge (x_1, r) \in R_{1,new}\}\| \\ R_2 := \{(a, x_2, r) \mid \exists x.(r, a, x) \in R_{new} \wedge (x_2, r) \in R_{2,new}\}\| \\ R := \{(r, a) \mid \exists x.(r, a, x) \in R_{new}\}\end{gathered}$$

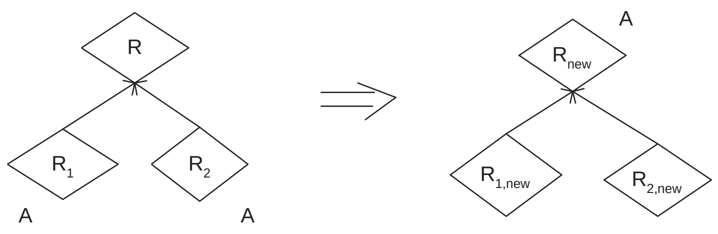

Figure 6.2: The relationship types before and after application of Rule 9

respectively.

Example 6.9. *An example is shown in Figure 6.2.*

□

Rule 10. For $comp(R_i) = \{r_i : R\}$ $(i = 1, \dots, n)$ and $A \in attr(R) - key(R)$ together with the constraint $\forall t \in R.\exists t' \in R_i.t'[r_i] = t$ replace the types such that $attr(R_{new}) = attr(R) - \{A\}$, $comp(R_{i,new}) = \{r_i : R_{new}\}$ and $attr(R_{i,new}) = attr(R_i) \cup \{A\}$ hold.

$$\frac{\begin{array}{rl} \mathfrak{M} \rhd & R = r : id \times A : t_a \times t \rightarrow \{\mathbb{1}\} \\ & R_1 = r : ref \times t_1 \rightarrow \{\mathbb{1}\} \\ & \dots \\ & R_n = r : ref \times t_n \rightarrow \{\mathbb{1}\} \end{array}}{\begin{array}{rl} \mathfrak{M}^* \rhd & R_{new} = r_{new} : id \times t \rightarrow \{\mathbb{1}\} \\ & R_{1,new} = r_{new} : ref \times t_1 \times A : t_a \rightarrow \{\mathbb{1}\} \\ & \dots \\ & R_{n,new} = r_{new} : ref \times t_n \times A : t_a \rightarrow \{\mathbb{1}\} \end{array}} \varphi$$

with the side condition φ being:

$A \in attr(R) - key(R)$ and $\forall t \in R.\exists t' \in R_i.t'[r] = t[r]$

The corresponding predicate $\mathcal{A}$:

$$\begin{array}{l} \forall x, a, x_1, \dots, x_n \\ \quad ((\exists r.R_{new}(r, x) = 1 \wedge R_{1,new}(r, x_1, a) = 1 \\ \qquad \Leftrightarrow \exists r.(R(r, a, x) = 1 \wedge R_1(r, x_1) = 1) \wedge \dots \wedge \\ \quad (\exists r.R_{new}(r, x) = 1 \wedge R_{n,new}(r, x_n, a) = 1 \\ \qquad \Leftrightarrow \exists r.(R(r, a, x) = 1 \wedge R_n(r, x_n) = 1)) \end{array}$$

The corresponding computable queries f and g:

$$\begin{array}{rl} R_{new} & := \{(r, x) \mid \exists a.(r, a, x) \in R\} \| \\ R_{1,new} & := \{(r, x_1, a) \mid (r, x_1) \in R_1 \wedge \exists x.(r, a, x) \in R\} \| \dots \| \\ R_{n,new} & := \{(r, x_n, a) \mid (r, x_n) \in R_n \wedge \exists x.(r, a, x) \in R\} \end{array}$$

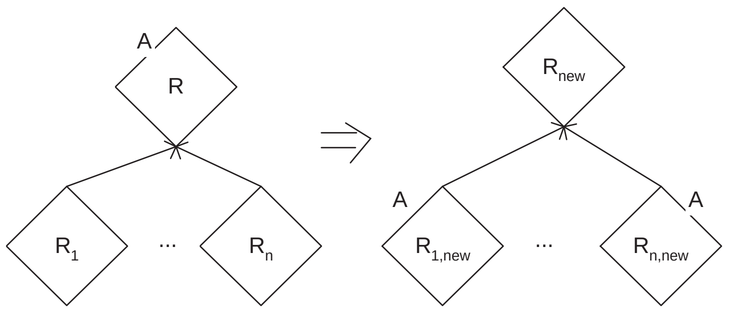

Figure 6.3: The relationship types before and after application of Rule 10

and

$$\begin{array}{rll} R := & \{(r,a,x) & \mid (r,x) \in R_{new} \wedge \exists x_1.(r,x_1,a) \in R_{1,new} \\ & & \wedge \cdots \wedge \exists x_n.(r,x_n,a) \in R_{n,new}\} \| \\ R_1 := & \{(r,x_1) & \mid \exists a.(r,x_1,a) \in R_{1,new}\} \| \ldots \| \\ R_n := & \{(r,x_n) & \mid \exists a.(r,x_n,a) \in R_{n,new}\} \end{array}$$

respectively.

Example 6.10. *An example is shown in Figure 6.3.*

□

The next two rules Rule 11 and 12 concern the reorganisation of paths and the shifting of attributes along paths. In both cases we obtain a dominating schema. Rule 11 could be split into two rules dealing separately with binary and unary relationship types R_n.

Rule 11. For a path $P \equiv R_1 - \cdots - R_n$ and a relationship type R with $r_n : R_n \in comp(R)$ together with path cardinality constraints $card(P, R_1) \leq (1,1) \leq card(P, R_n)$ replace R such that $comp(R_{new}) = comp(R) - \{r_n : R_n\} \cup \{r_{1,new} : R_1\}$ with a new role $r_{1,new}$ holds.

$$\frac{\begin{array}{ll} \mathfrak{M} \rhd & R = r_n : ref \times t \to \{\mathbb{1}\} \\ & R_1 = r_1 : id \times [r_2 : ref] \times t_1 \to \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \to \{\mathbb{1}\} \\ & \ldots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times t_n \to \{\mathbb{1}\} \end{array}}{\begin{array}{ll} \mathfrak{M}^* \rhd & R_{new} = r_1 : ref \times t \to \{\mathbb{1}\} \\ & R_1 = r_1 : id \times [r_2 : ref] \times t_1 \to \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \to \{\mathbb{1}\} \\ & \ldots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times t_n \to \{\mathbb{1}\} \end{array}} \varphi$$

where $[r_i : ref](1 \leq i \leq n)$ models a possible reference to type R_i, due to the path definition $P \equiv R_1 - \cdots - R_n$ covering all possible path directions, as depicted in Figure 6.4. We say that $[r_i : ref]$ is realised, denoted by $[r_{i+1} : ref] \equiv r_{i+1} : ref$ iff $[r_i : ref]$ models an actual reference to R_i.

The side condition φ:

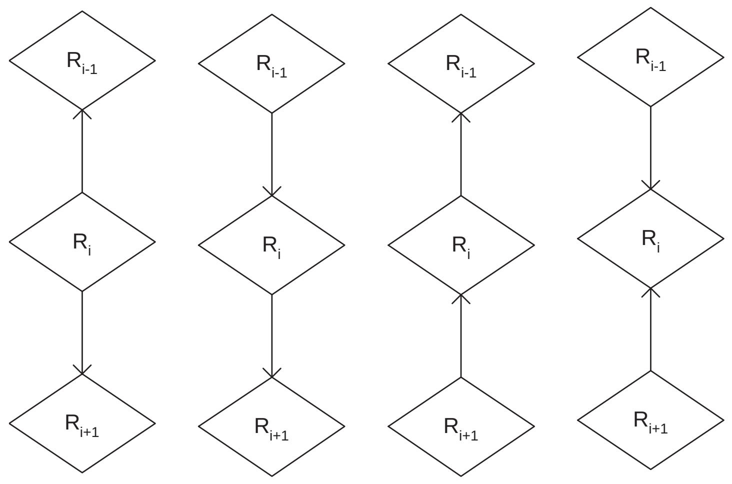

Figure 6.4: The possible path directions

- $card(P, R_1) \leq (1,1) \leq card(P, R_n)$, and
- $P \equiv R_1 - \cdots - R_n$ holds, which means:

  $$\forall i(1 \leq i < n).([r_{i+1} : ref] \equiv r_{i+1} : ref \text{ in } R_i) \text{ xor } ([r_i : ref] \equiv r_i : ref \text{ in } R_{i+1})$$

The corresponding predicate $\mathcal{A}$:

$$\begin{aligned}&\forall r_n, x.(R(r_n, x) = 1 \Leftrightarrow \\ &\quad \exists r_1, \ldots r_{n-1}, x_1, \ldots, x_n.(R_{new}(r_1, x) = 1) \wedge R_1(\pi_{T_1}(r_1, r_2, x_1)) = 1 \wedge \\ &\qquad R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, x_n)) = 1)\end{aligned}$$

where T_i is the source type of R_i, i.e. we have $R_i = T_i \to \{\mathbb{1}\}$. $\pi_{T_i}(r_i, r_{i-1}, r_{i+1}, x_i)$ projects the tuple $(r_i, r_{i-1}, r_{i+1}, x_i)$ onto T_i.

The corresponding computable queries f and g:

$$\begin{aligned}R_{new} := \{&(h(S_{r_1}), x) \mid S_{r_1} = \{r_1 \mid \exists r_2, \ldots, r_n, x_1, \ldots, x_n.(\pi_{T_1}(r_1, r_2, x_1) \in R_1 \wedge \\ &\pi_{T_2}(r_2, r_1, r_3, x_2) \in R_2 \wedge \cdots \wedge \pi_{T_n}(r_n, r_{n-1}, x_n) \in R_n \wedge (r_n, x) \in R)\} \neq \emptyset\}\end{aligned}$$

and

$$\begin{aligned}R := \{&(r_n, x) \mid \exists r_1, \ldots r_{n-1}, x_1, \ldots, x_{n-1}.((r_1, x) \in R_{new} \wedge \pi_{T_1}(r_1, r_2, x_1) \in R_1 \wedge \\ &\pi_{T_2}(r_2, r_1, r_3, x_2) \in R_2 \wedge \cdots \wedge \pi_{T_n}(r_n, r_{n-1}, x_n) \in R_n)\}\end{aligned}$$

respectively, for some fixed computable selection function h with the property $h(S) \in S$.

Example 6.11. *An example is shown in Figure 6.5.*

□

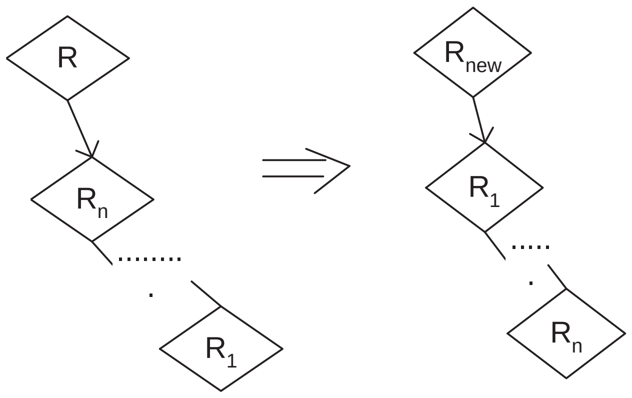

Figure 6.5: The relationship types before and after application of Rule 11

Rule 12. For a path $P \equiv R_1 - \cdots - R_n$ with $A \in attr(R_n)$, $A \notin key(R_n)$, and path cardinality constraints $card(P, R_1) \leq (1,1) \leq card(P, R_n)$ replace R_1, R_n such that $attr(R_{1,new}) = attr(R_1) \cup \{A\}$ and $attr(R_{n,new}) = attr(R_n) - \{A\}$ hold.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R_1 = r_1 : id \times [r_2 : ref] \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \rightarrow \{\mathbb{1}\} \\ & \ldots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times A : t_a \times t_n \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & R_{1,new} = r_{1,new} : id \times [r_2 : ref] \times A : t_a \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \rightarrow \{\mathbb{1}\} \\ & \ldots \\ & R_{n,new} = r_{n,new} : id \times t_n \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

where $[r_i : ref](1 \leq i \leq n)$ models a possible reference to type R_i, due to the path definition $P \equiv R_1 - \cdots - R_n$ covering all possible path directions, as depicted in Figure 6.4. We say that $[r_i : ref]$ is realised, denoted by $[r_{i+1} : ref] \equiv r_{i+1} : ref$ iff $[r_i : ref]$ models an actual reference to R_i.

The side condition φ:

- $A \notin key(R_n)$, and
- $card(P, R_1) = (1,1) \leq card(P, R_n)$, and
- $P \equiv R_1 - \cdots - R_n$,which means:

  $\forall i (1 \leq i < n).([r_{i+1} : ref] \equiv r_{i+1} : ref$ in $R_i)$ xor $([r_i : ref] \equiv r_i : ref$ in $R_{i+1})$;

The corresponding predicate $\mathcal{A}$:

$$\begin{array}{l}\forall r_1, \ldots, r_n, a, x_1, \ldots, x_n. \\ \quad ([R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_{n-1}(\pi_{T_{n-1}}(r_{n-1}, r_{n-2}, r_n, x_n)) = 1] \Rightarrow\end{array}$$

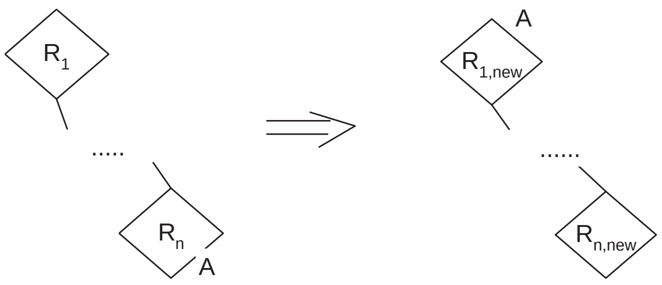

Figure 6.6: The relationship types before and after application of Rule 12

$$[R_1(\pi_{T_1}(r_1, r_2, x_1)) = 1 \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, a, x_n)) = 1 \Leftrightarrow$$
$$R_{1,new}(\pi_{T_{1,new}}(r_1, r_2, a, x_1)) = 1 \wedge R_{n,new}(\pi_{T_{n,new}}(r_n, r_{n-1}, x_n)) = 1])$$

where T_i is the source type of R_i, i.e. we have $R_i = T_i \rightarrow \{\mathbb{1}\}$. $\pi_{T_i}(r_i, r_{i-1}, r_{i+1}, x_i)$ projects the tuple $(r_i, r_{i-1}, r_{i+1}, x_i)$ onto T_i.

The corresponding computable queries f and g:

$$\begin{aligned}
R_{n,new} &:= \{\pi_{T_{n,new}}(r_n, r_{n-1}, x_n) \mid \exists a.\pi_{T_n}(r_n, r_{n-1}, a, x_n) \in R_n\}\| \\
R_{1,new} &:= \{\pi_{T_{1,new}}(r_1, r_2, a, x_1) \mid \exists r_3, \ldots, r_n, x_2, \ldots, x_n. \\
&\quad (\pi_{T_1}(r_1, r_2, x_1) \in R_1 \wedge \pi_{T_2}(r_2, r_1, r_3, x_2) \in R_2 \wedge \cdots \wedge \\
&\quad\quad \pi_{T_n}(r_n, r_{n-1}, a, x_n) \in R_n\}
\end{aligned}$$

and

$$\begin{aligned}
R_n &:= \{\pi_{T_n}(r_n, r_{n-1}, a, x_n) \mid \exists r_1, r_2, \ldots, r_{n-2}, x_1, \ldots, x_{n-1}. \\
&\quad (\pi_{T_{1,new}}(r_1, r_2, a, x_1) \in R_{1,new} \wedge \pi_{T_2}(r_2, r_1, r_3, x_2) \in R_2 \wedge \cdots \wedge \\
&\quad \pi_{T_{n,new}}(r_n, r_{n-1}, x_n) \in R_{n,new}\}\| \\
R_1 &:= \{\pi_{T_1}(r_1, r_2, x_1) \mid \exists a.\pi_{T_{1,new}}(r_1, r_2, a, x_1) \in R_{1,new}\}
\end{aligned}$$

respectively.

Example 6.12. *An example is shown in Figure 6.6.*

□

6.5.3 Schema Extension

The third group of rules deal with the schema extensions. This either concerns new attributes, new types, new subtypes or the simplification of hierarchies. These rules are needed in step 1 of our method.

Rule 13. Add a new type R. In such case we obtain a dominant schema.

$$\mathfrak{M}^* \ \triangleright R = \ell : label \times t$$

With a new type added we get a dominant schema. The corresponding abstraction predicate $\mathcal{A}$ is simply defined as *true*. The corresponding computable queries f and g can

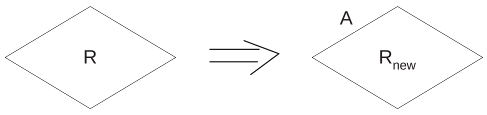

Figure 6.7: The relationship types before and after application of Rule 14

be defined as identity function.

Rule 14. Add a new attribute A to the type R, i.e. $attr(R_{new}) = attr(R) \cup \{A\}$. In addition, the new attribute may be used to extend the key, i.e. we may have $key(R_{new}) = key(R) \cup \{A\}$.

$$\frac{\mathfrak{M} \rhd R = t \to \{\mathbb{1}\}}{\mathfrak{M}^* \ \rhd R_{new} = A : t_a \times t \to \{\mathbb{1}\}}$$

with no side condition.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x.(\exists a.R_{new}(a,x) = 1 \Leftrightarrow R(x) = 1)$$

The corresponding computable queries f and g:

$$R_{new} := \{(a,x) \mid (x) \in R\}, \text{ for a constant } a$$

and

$$R := \{(x) \mid \exists a.(x,a) \in R_{new}\}$$

respectively.

Example 6.13. *An example is shown in Figure 6.7.*

□

Adding a new attribute A by Rule 14 does not change the cardinality of the type R, i.e. $card(R) = card(R_{new})$. This rule always results in a dominant schema.

The next two rules allow to introduce a new subtype via selection or projection on non-key-attributes. In both cases we have schema equivalence.

Rule 15. For a type R introduce a new relationship type R_{new} with $comp(R_{new}) = \{r : R\} = key(Rnew)$ and add a constraint $R_{new} = \sigma_{\varphi}(R)$ for some selection formula φ.

$$\frac{\mathfrak{M} \rhd \quad R = r : id \times t \to \{\mathbb{1}\}}{\begin{array}{rl}\mathfrak{M}^* \rhd & R = r : id \times t \to \{\mathbb{1}\} \\ & R_{new} = r : ref \to \{\mathbb{1}\}\end{array}}$$

with a constraint:

$$R_{new} = \sigma_\varphi(R)$$

for some selection formula φ.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall r.(R_{new}(r) = 1 \Leftrightarrow \exists x.\varphi(x) = 1 \wedge R(r, x) = 1)$$

The corresponding computable queries f:

$$R_{new} := \{(r) \mid \exists x.(\varphi(x) = 1 \wedge (r, x) \in R)\}$$

and g an identity function, respectively.

Example 6.14. *Let us look at the loan application again. We may use this rule to create a type called* ELDERLY *from the type* CUSTOMER *with φ being date_of_birth $\leq$ 1940, which can be used in speeding up the process when* CUSTOMER *is big.*

□

Rule 16. For a type R and attributes $A_1, \ldots, A_n \in attr(R)$ such that there are no $B_i \in key(R)$ with $A_i \geq B_i$ (for projection on non-key-attributes) introduce a new relationship type R_{new} with $comp(R_{new}) = \{r : R\} = key(R_{new})$ and $attr(R_{new}) = \{A_1, \ldots, A_n\}$, and add a constraint $R_{new} = \pi_{A_1,\ldots,A_n}(R)$.

$$\frac{\mathfrak{M} \rhd \quad R = r : id \times A_1 : t_1 \times \ldots A_n : t_n \times t \to \{\mathbb{1}\}}{\begin{array}{rl}\mathfrak{M}^* \rhd & R = r : id \times A_1 : t_1 \times \ldots A_n : t_n \times t \to \{\mathbb{1}\} \\ & R_{new} = r : ref \times A_1 : t_1 \times \ldots A_n : t_n \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$\forall B_i \in key(R) \text{ we have } A_i \not\geq B_i.$$

The constraint:

$$key(R_{new}) = \{r : ref\}$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x_1, \ldots, x_n, r.((R_{new}(r, x_1, \ldots, x_n) = 1) \Leftrightarrow \exists x.R(r, x_1, \ldots, x_n, x) = 1)$$

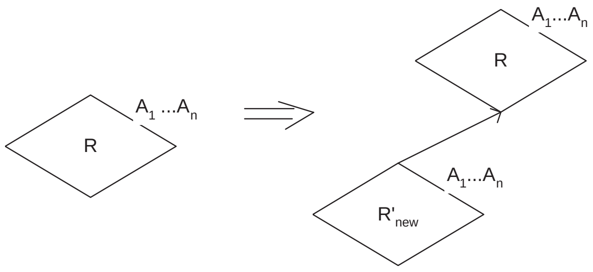

Figure 6.8: The relationship types before and after application of Rule 16

The corresponding computable queries f:

$$R_{new} := \{(r, x_1, \ldots x_n) \mid \exists x.(r, x_1, \ldots x_n, x) \in R\}$$

and g an identity function, respectively.

Example 6.15. *An example is shown in Figure 6.8.*

□

The last rule 17 in this group allows to simplify hierarchies.

Rule 17. Replace types R, $R_1, \ldots, R_n$ with $comp(R_i) = \{r_i : R\} = key(R_i)$ and $card(R_i, R) = (1,1)$ $(i = 1, \ldots, n)$ by a new type R_{new} with $comp(R_{new}) = comp(R)$, $attr(R_{new}) = attr(R) \cup \bigcup_{i=1}^{n} attr(R_i)$ and $key(R_{new}) = key(R)$.

$$\frac{\begin{array}{ll} \mathfrak{M} \rhd & R = r : id \times t \to \{\mathbb{1}\} \\ & R_1 = r : ref \times t_1 \to \{\mathbb{1}\} \\ & \ldots \\ & R_n = r : ref \times t_n \to \{\mathbb{1}\} \end{array}}{\mathfrak{M}^* \rhd \quad R_{new} = r : id \times t \times t_1 \cdots \times t_n \to \{\mathbb{1}\}} \varphi$$

with the side condition φ:

$$comp(R_i) = \{r_i : R\} = key(R_i) \wedge card(R, R_i) = (1,1)(i = 1, \ldots, n)$$

The constraint:

$$key(R_{new}) = key(R)$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall x_1, \ldots, x_n, r, x.(R'_{new}(r, x, x_1, \ldots, x_n) = 1 \Leftrightarrow \\ (R(r,x) = 1 \wedge R_1(r, x_1) = 1 \wedge \cdots \wedge R_n(r, x_n) = 1))$$

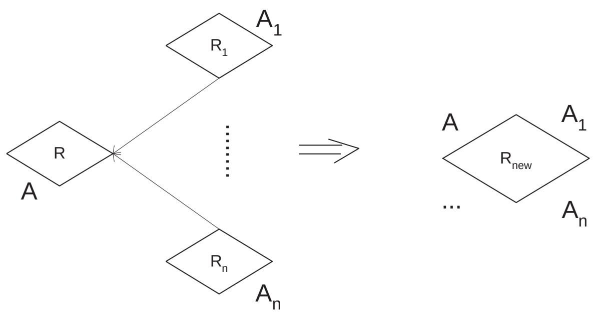

Figure 6.9: The relationship types before and after application of Rule 17

The corresponding computable queries f and g:

$$R_{new} := \{(r, x, x_1, \dots x_n) \mid (r, x) \in R \wedge (r, x_1) \in R_1 \wedge \dots \wedge (r, x_n) \in R_n\}$$

and

$$\begin{aligned} R &:= \{(r, x) \mid \exists x_1, \dots x_n.((r, x, x_1, \dots, x_n) \in R_{new})\} \| \\ R_1 &:= \{(r, x_1) \mid \exists x, x_2, \dots, x_n.(r, x, x_1, \dots, x_n) \in R_{new}\} \| \dots \| \\ R_n &:= \{(r, x_n) \mid \exists x, x_1, \dots, x_{n-1}.(r, x, x_1, \dots, x_n) \in R_{new}\} \end{aligned}$$

respectively.

Example 6.16. *An example is shown in Figure 6.9.*

□

6.5.4 Type Integration.

The fourth group of rules deals with the integration of types in step 4 of our method. Rule 18 considers the equality case, Rule 19 considers the containment case, and Rule 20 covers the overlap case. Note that these transformation rules cover the core of the approaches in [60, 104, 63].

Rule 18. If R_1 and R_2 are types with $key(R_1) = key(R_2)$ and we have the constraint $R_1[key(R_1) \cup X] = h(R_2[key(R_2) \cup Y])$ for some $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping h, then replace these types by R_{new} with $comp(R_{new}) = comp(R_1) \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{new}) = attr(R_1) \cup (attr(R_2) - Y - key(R_2)) \cup \{D\}$ and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D.

$$\frac{\begin{array}{rl} \mathfrak{M} \rhd & R_1 = K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ & R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \to \{\mathbb{1}\} \end{array}}{\mathfrak{M}^* \rhd \quad R_{new} = K_1 : t_{k_1} \times X : t_x \times t_1 \times t_2 \to \{\mathbb{1}\}} \varphi$$

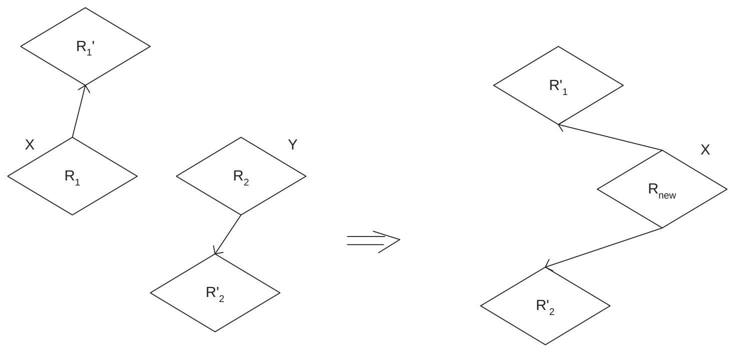

Figure 6.10: The relationship types before and after application of Rule 18

with the side condition φ:

$$K_1 = K_2 \wedge R_1[K_1 \cup X] = h(R_2[K_2 \cup Y]),$$

where K_1 and K_2 are the keys, h is a fixed computable bijective mapping.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall k_1, x, x_1, x_2.(R_{new}(k_1, x, x_1, x_2) = 1$$
$$\Leftrightarrow R_1(k_1, x, x_1) = 1 \wedge R_2(h^{-1}(k_1, x), x_2) = 1)$$

The corresponding computable queries f and g:

$$R_{new} := \{(k_1, x, x_1, x_2) \mid (k_1, x, x_1) \in R_1 \wedge (h^{-1}(k_1, x), x_2) \in R_2\}$$

and

$$R_1 := \{(k_1, x, x_1) \mid \exists x_2.(k_1, x, x_1, x_2) \in R_{new}\} \|$$
$$R_2 := \{(h^{-1}(k_1, x), x_2) \mid \exists x_1.(k_1, x, x_1, x_2) \in R_{new}\}$$

respectively.

Example 6.17. *An example is shown in Figure 6.10.*

□

When X and Y are empty, then Rule 18 merges two types by combining the two attribute sets.

Rule 19. If R_1 and R_2 are types with $key(R_1) = key(R_2)$ and the constraint $R_2[key(R_2) \cup Y] \subset h(R_1[key(R_1) \cup X]$ holds for some $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping h, then replace R_1 by $R_{1,new}$ with $comp(R_{1,new}) = comp(R_1)$, $attr(R_{new}) = attr(R_1) \cup \{D\}$ and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D. Furthermore, replace R_2 by $R_{2,new}$ with $comp(R_{2,new}) = \{r_{new} : R_{1,new}\} \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{2,new}) = attr(R_2) - Y - key(R_2)$ and $key(R_{2,new}) = \{r_{new} : R_{1,new}\}$.

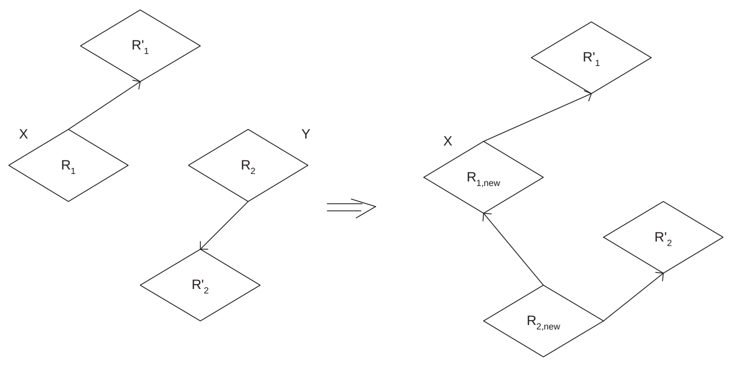

Figure 6.11: The relationship types before and after application of Rule 19

$$\frac{\begin{array}{rl}\mathfrak{M} \rhd & R_1 = r_1 : id \times K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ & R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \to \{\mathbb{1}\}\end{array}}{\begin{array}{rl}\mathfrak{M}^* \rhd & R_1 = r_1 : id \times K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ & R_{2,new} = r_1 : ref \times t_2 \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$K_1 = K_2 \wedge R_2[K_2 \cup X] \subseteq h(R_1[K_1 \cup Y]),$$

where K_1 and K_2 are the keys, h is a fixed computable bijective mapping.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall k, x, x_1, x_2, r.(R_1(r, k, x, x_1) = 1 \Rightarrow$$
$$(R_2(h(k, x), x_2) = 1 \Leftrightarrow R_{2,new}(r, x_2) = 1))$$

The corresponding computable queries f and g:

$$R_{2,new} := \{(r, x_2) \mid \exists k, x.((k, x, x_2) \in R_2 \wedge \exists x_1.(r, h(k, x), x_1) \in R_1\}$$

and

$$R_2 := \{(k, x, x_2) \mid \exists r.((r, x_2) \in R_{2,new} \wedge \exists x_1.(r, h(k, x), x_1) \in R_{1,new})\}$$

respectively.

Example 6.18. *An example is shown in Figure 6.11.*

□

Rule 20. If R_1 and R_2 are types with $key(R_1) = key(R_2)$ such that for $X \subseteq comp(R_1) \cup attr(R_1)$, $Y \subseteq comp(R_2) \cup attr(R_2)$ and a bijective mapping h the constraints

$$\begin{array}{ll} R_2[key(R_2) \cup Y] \subseteq h(R_1[key(R_1) \cup X]) & , \\ R_2[key(R_2) \cup Y] \supseteq h(R_1[key(R_1) \cup X]) & \text{and} \\ R_2[key(R_2) \cup Y] \cap h(R_1[key(R_1) \cup X]) = \emptyset & \end{array}$$

are not satisfied (the first two cases are covered by Rule 18 and 19, the last one has no case for integration) then replace R_1 by $R_{1,new}$ with $comp(R_{1,new}) = \{r_{1,new} : R_{new}\} \cup (comp(R_1) - X - key(R_1))$, $attr(R_{1,new}) = attr(R_1) - X - key(R_1)$ and $key(R_{1,new}) = \{r_{1,new} : R_{new}\}$, replace R_2 by $R_{2,new}$ with $comp(R_{2,new}) = \{r_{new} : R_{1,new}\} \cup (comp(R_2) - Y - key(R_2))$, $attr(R_{2,new}) = attr(R_2) - Y - key(R_2)$ and $key(R_{2,new}) = \{r_{new} : R_{1,new}\}$ and introduce a new type R_{new} with $comp(R_{new}) = comp(R_1) \cap (key(R_1) \cup X)$, $attr(R_{new}) = attr(R_1) \cap (X \cup key(R_1) \cup \{D\}$, and $key(R_{new}) = key(R_1) \cup \{D\}$ and an optional new distinguishing attribute D.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R_1 = K_1 : t_{k_1} \times X : t_x \times t_1 \to \{\mathbb{1}\} \\ & R_2 = K_2 : t_{k_2} \times Y : t_y \times t_2 \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & R_{1,new} = r : ref \times t_1 \to \{\mathbb{1}\} \\ & R_{2,new} = r : ref \times t_2 \to \{\mathbb{1}\} \\ & R_{new} = r : id \times K_1 : t_{k_1} \times X : t_x \to \{\mathbb{1}\}\end{array}} \varphi$$

with the side condition φ:

$$K_1 = K_2$$

where K_1 and K_2 are the keys.

The guideline: apply the rule when none of the following hold:

$$\begin{array}{l} R_2[key(R_2) \cup Y] \subseteq h(R_1[key(R_1) \cup X]), \\ R_2[key(R_2) \cup Y] \supseteq h(R_1[key(R_1) \cup X]) \quad \text{and} \\ R_2[key(R_2) \cup Y] \cap h(R_1[key(R_1) \cup X]) = \emptyset \end{array}$$

with a fixed computable bijective mapping h.

The corresponding abstraction predicate $\mathcal{A}$:
$\forall k, x, x_1, x_2.$
$((\exists r.(R_{new}(r,k,x) = 1 \wedge R_{1,new}(r,x_1) = 1) \Leftrightarrow R_1(k,x,x_1) = 1) \wedge$
$(\exists r.(R_{new}(r,k,x) = 1 \wedge R_{2,new}(r,x_2) = 1) \Leftrightarrow R_2(h^{-1}(k,x),x_2) = 1))$

The corresponding computable queries f and g:

$$\begin{array}{r} R_{1,new} := \{(z(k),x_1) \mid \exists x.(k,x,x_1) \in R_1\| \\ R_{2,new} := \{(z(k),x_2) \mid \exists x.(h(k,x),x_2) \in R_2\| \\ R_{new} := \{(z(k),k,x) \mid \exists x_1.(k,x,x_1) \in R_1 \vee \exists x_2.(h(k,x),x_2) \in R_2\} \end{array}$$

and

$$\begin{array}{r} R_1 := \{(k,x,x_1) \mid \exists r.((r,k,x) \in R_{new} \wedge (r,x_1) \in R_{1,new})\| \\ R_2 := \{(h(k,x),x_2) \mid \exists r.((r,k,x) \in R_{new} \wedge (r,x_2) \in R_{2,new})\} \end{array}$$

respectively, for some fixed computable injective function $z : K_1 \to id$.

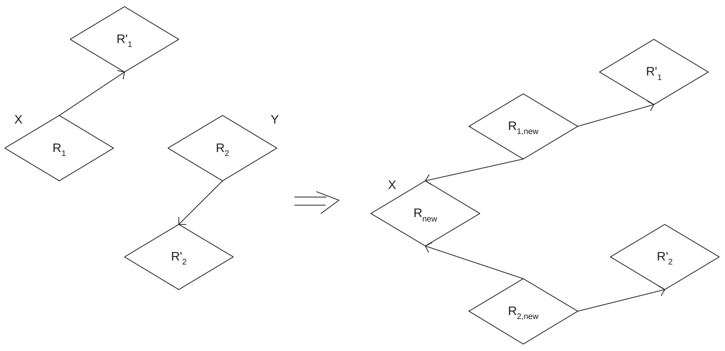

Figure 6.12: The relationship types before and after application of Rule 20

Example 6.19. *An example is shown in Figure 6.12.*

□

Rule18-20 could each be split into several rules depending on f being the identity or not and the necessity to introduce D or not. In all cases we obtain dominance. Rule 21 considers the case of a selection condition, in which schema equivalence holds.

Rule 21. If R and R' are types with $comp(R') \cup attr(R') = Z \subseteq comp(R) \cup attr(R)$ such that the constraint $R' = \sigma_\varphi(\pi_Z(R))$ holds for some selection condition φ, then omit R'.

$$\frac{\begin{array}{rl}\mathfrak{M} \rhd & R = Z : t_z \times t \to \{\mathbb{1}\} \\ & R' = Z : t_z \to \{\mathbb{1}\}\end{array}}{\mathfrak{M}^* \rhd \quad R = Z : t_z \times t \to \{\mathbb{1}\}} \ \varphi$$

with side condition φ:

$$R' = \sigma_\psi(\pi_Z(R))$$

for some selection condition ψ.

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall z.(R'(z) = 1 \Leftrightarrow \exists x.R(z,x) = 1 \wedge \psi(z) = 1)$$

The corresponding computable queries g:

$$R' := \{(z) \mid \exists x.(z,x) \in R_1 \wedge \psi(z) = 1\}$$

and f an identity function.

Example 6.20. *An example is shown in Figure 6.13.*

□

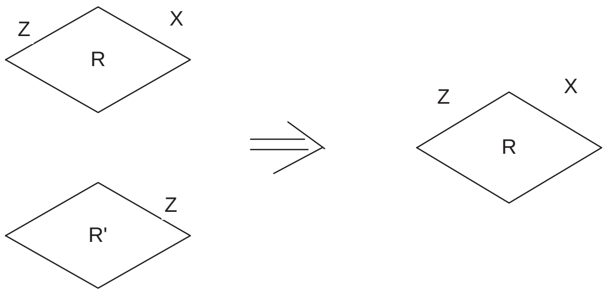

Figure 6.13: The relationship types before and after application of Rule 21

6.5.5 Handling Integrity Constraints.

The fifth group of rules to be applied in step 5 of our method concerns transformations originating from path inclusion constraints. Rule 22 allows us to change a relationship type. This rule leads to equivalent schemata. Rule 23 allows to introduce a relationship type and a join dependency. Finally, Rule 24 handles a condition under which a relationship type may be omitted. Both Rule 23 and Rule 24 guarantee dominance.

Rule 22. If there are paths $P \equiv R - R_1$ and $P' \equiv R' - R_1$ with $\{r_1 : R_1\} \in \mathit{comp}(R)$ and $\{r_1' : R_1\} \in \mathit{comp}(R')$ such that the constraint $P[R_1] \subseteq P'[R_1]$ holds, then replace R in such a way that $\mathit{comp}(R_{new}) = \mathit{comp}(R) - \{r_1 : R_1\} \vee \{r_{new} : R'\}$, $\mathit{attr}(R_{new}) = \mathit{attr}(R)$ and $\text{key}(R_{new}) = \text{key}(R) - \{r_1 : R_1\} \cup \{r_{new} : R'\}$ if $\{r_1 : R_1\} \in \text{key}(R), \text{key}(R_{new}) = \text{key}(R)$, otherwise.

$$\frac{\begin{array}{ll}\mathfrak{M} \rhd & R_1 = r_1 : id \times t_1 \rightarrow \{\mathbb{1}\} \\ & R = r_1 : ref \times t \rightarrow \{\mathbb{1}\} \\ & R' = r' : id \times r_1 : ref \times t' \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \rhd & R_1 = r_1 : id \times t_1 \rightarrow \{\mathbb{1}\} \\ & R' = r' : id \times r_1 : ref \times t' \rightarrow \{\mathbb{1}\} \\ & R_{new} = r' : ref \times t \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

with side condition φ:

$$P[R_1] \subseteq P'[R_1]$$

The constraints:

$$\text{key}(R_{new}) = \begin{cases} \text{key}(R) - \{r_1 : R_1\} \cup \{r' : R'\} & \text{if } \{r_1 : R_1\} \in \text{key}(R) \\ \text{key}(R) & \text{otherwise} \end{cases}$$

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall r_1, x.(R(r_1, x) = 1 \Leftrightarrow \\ \exists r', x'.(R_{new}(r', x) = 1 \wedge R'(r', r_1, x') = 1))$$

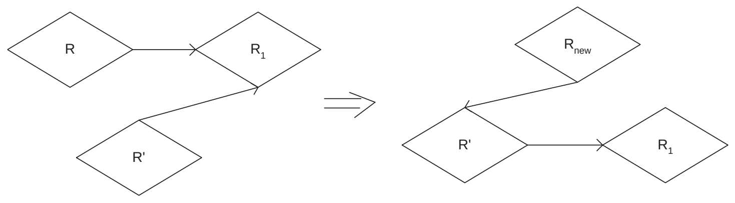

Figure 6.14: The relationship types before and after application of Rule 22

The corresponding computable queries f and g:

$$R_{new} := \{(h(S_{r'}), x) \mid S_{r'} = \{r' \mid \exists r_1.((r_1, x) \in R \wedge \exists x'.(r', r_1, x') \in R')\} \neq \emptyset\}$$

and

$$R := \{(r_1, x) \mid \exists r'.((r', x) \in R_{new} \wedge \exists x'.(r', r_1, x') \in R')\}$$

respectively, where h is a fixed computable function with the property $h(S) \in S$.

Example 6.21. *An example is shown in Figure 6.14.*

$\square$

Rule 23. If there are paths $P \equiv R - R_1$ and $P' \equiv R' - R_1$ with $\{r_1 : R_1\} \in comp(R)$ and $\{r_1' : R_1\} \in comp(R')$ such that the constraint $P[R_1] = P'[R_1]$ holds, then replace R and R' by R_{new} such that $comp(R_{new}) = (comp(R) - \{r_1 : R_1\}) \cup (comp(R') - \{r_1' : R_1\}) \cup \{r_{1,new} : R_1\}$, $attr(R_{new}) = attr(R) \cup attr(R')$ and $key(R_{new}) = (key(R) - \{r_1 : R_1\}) \cup (key(R') - \{r_1' : R_1\}) \cup \{r_{1,new} : R_1\}$, if $\{r_1 : R_1\} \in key(R)$ or $\{r_1' : R_1\} \in \cup key(R')$, or $key(R_{new}) = key(R) \cup key(R')$ hold. Add the join dependency $R_{new}[X, r_{1,new}] \bowtie R_{new}[r_{1,new}, Y] \subseteq R_{new}[X, r_{1,new}, Y]$, where $X = attr(R)$ and $Y = attr(R')$.

$$\frac{\begin{array}{ll} \mathfrak{M} \rhd & R_1 = r_1 : id \times t_1 \to \{\mathbb{1}\} \\ & R = r_1 : ref \times t \to \{\mathbb{1}\} \\ & R' = r_1 : ref \times t' \to \{\mathbb{1}\} \end{array}}{\begin{array}{ll} \mathfrak{M}^* \rhd & R_1 = r_1 : id \times t_1 \to \{\mathbb{1}\} \\ & R_{new} = r_1 : ref \times t \times t' \to \{\mathbb{1}\} \end{array}} \varphi$$

with side condition φ:

$$P[R_1] = P'[R_1]$$

The constraint:

$$R_{new}[t, r_{1,new}] \bowtie R_{new}[r_{1,new}, t'] \subseteq R_{new}[t, r_{1,new}, t'], \text{ and}$$

$$\text{key}(R_{new}) = \begin{cases} (key(R) - \{r_1 : R_1\}) \cup (key(R') - \{r_1 : R_1\}) \cup \{r_{1,new} : R_1\} \\ \qquad \text{if } \{r_1 : R_1\} \in \text{key}(R) \cup \text{key}(R') \\ \text{key}(R) \quad \text{otherwise} \end{cases}$$

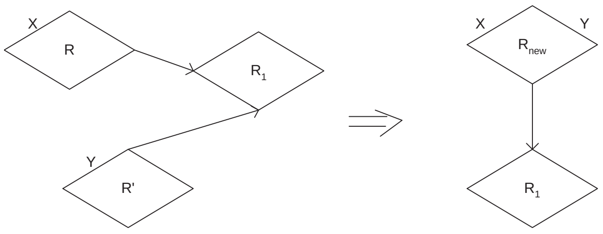

Figure 6.15: The relationship types before and after application of Rule 23

The corresponding abstraction predicate $\mathcal{A}$:

$$\forall r_1, x, x' (R_{new}(r_1, x, x') = 1 \Leftrightarrow R(r_1, x) = 1 \wedge R(r_1, x') = 1)$$

The corresponding computable queries f and g:

$$R_{new} := \{(r_1, x, x') \mid (r_1, x) \in R \wedge (r_1, x') \in R'\}$$

and

$$\begin{array}{l} R := \{(r_1, x) \mid \exists x'.(r_1, x, x') \in R_{new}\} \| \\ R' := \{(r_1, x') \mid \exists x.(r_1, x, x') \in R_{new}\} \end{array}$$

respectively.

Example 6.22. *An example is shown in Figure 6.15.*

$\square$

Rule 24. If there are paths $P \equiv R_1 - R_2 - \cdots - R_n$ and $P' \equiv R_1 - R - R_n$ with $comp(R) = \{r_1 : R_1, r_n : R_n\}$ and $attr(R) = \emptyset$ such that the constraint $P[R_1, R_n] = P'[R_1, R_n]$ holds, then omit R.

$$\frac{\begin{array}{ll} \mathfrak{M} \rhd & R_1 = r_1 : id \times [r_2 : ref] \times t_1 \to \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \to \{\mathbb{1}\} \\ & \ldots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times t_n \to \{\mathbb{1}\} \\ & R = r_1 : ref \times r_2 : ref \to \{\mathbb{1}\} \end{array}}{\begin{array}{ll} \mathfrak{M}^* \rhd & R_1 = r_1 : id \times [r_2 : ref] \times t_1 \to \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \to \{\mathbb{1}\} \\ & \ldots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times t_n \to \{\mathbb{1}\} \end{array}} \varphi$$

where $[r_i : ref](1 \leq i \leq n)$ models a possible reference to type R_i, due to the path definition $P \equiv R_1 - \cdots - R_n$ covering all possible path directions, as depicted in Figure 6.4. We say that $[r_i : ref]$ is realised, denoted by $[r_{i+1} : ref] \equiv r_{i+1} : ref$ iff $[r_i : ref]$ models an actual reference to R_i.

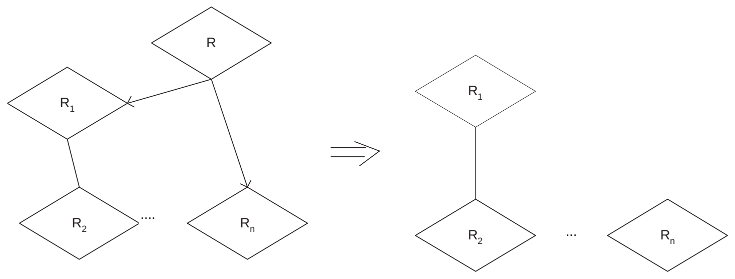

Figure 6.16: The relationship types before and after application of Rule 24

The side condition φ:

$$P[R_1, R_n] = P'[R_1, R_n]$$

The corresponding predicate $\mathcal{A}$:

$$\begin{aligned}&\forall r_1, r_n.(R(r_1, r_n) = 1 \Leftrightarrow\\ &\quad \exists r_2, \dots r_{n-1}, x_1, \dots, x_n.(R_1(\pi_{T_1}(r_1, r_2, x_1)) = 1 \wedge\\ &\qquad R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, x_n)) = 1)\end{aligned}$$

where T_i is the source type of R_i, i.e. we have $R_i = T_i \to \{1\!\!1\}$. $\pi_{T_i}(r_i, r_{i-1}, r_{i+1}, x_i)$ projects the tuple $(r_i, r_{i-1}, r_{i+1}, x_i)$ onto T_i.

The corresponding computable queries g:

$$\begin{aligned}R := \{(r_1, r_2) \mid \exists r_2, \dots r_{n-1}, x_1, \dots, x_n.(R_1(\pi_{T_1}(r_1, r_2, x_1)) = 1 \wedge\\ R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, x_n)) = 1)\}\end{aligned}$$

and f an identity function.

Example 6.23. *An example is shown in Figure 6.16.*

□

The final group of transformation rules Rule 25-28 permits to handle remaining constraints such as functional dependencies, path functional dependencies, and join dependencies. All these constraints are described in detail in [106]. The rules refer to step 6 of our method. Rule 25 handles vertical decomposition in the presence of a functional dependency. Rule 26 allows to simplify a key in the presence of a path functional dependency. Rule 27 introduces a new entity type in the presence of a path functional dependency. Finally, Rule 28 replaces a multi-ary relationship type by binary relationship types in the presence of a join dependency. The four rules lead to dominating schemata.

Rule 25. If a functional dependency $X \to A$ with a generalised subset X of $attr(E)$ and an attribute $A \in attr(E) - X$ holds on an entity type E, but $X \to key(E)$ does not hold, then remove A from $attr(E)$ and add a new entity type E'_{new} with $attr(E'_{new}) = X \cup \{A\}$ and $key(E'_{new}) = X$.

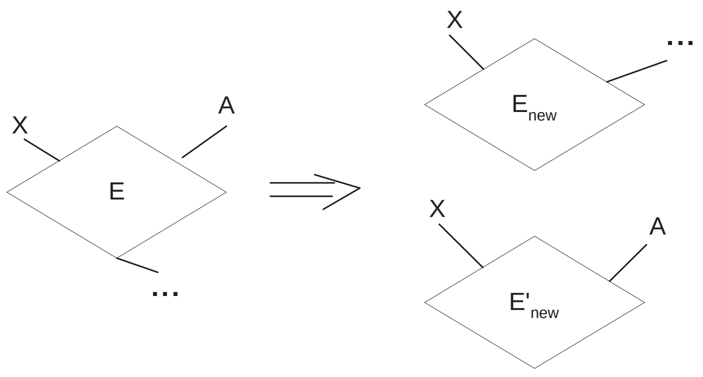

Figure 6.17: The relationship types before and after application of Rule 25

$$\frac{\mathfrak{M} \rhd \quad E = X : t_x \times A : t_a \times t \rightarrow \{\mathbb{1}\}}{\begin{array}{l}\mathfrak{M}^* \rhd \quad E_{new} = X : t_x \times t \rightarrow \{\mathbb{1}\} \\ \qquad\quad E'_{new} = X : t_x \times A : t_a \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

with side condition φ:

$$X \rightarrow A \text{ and } X \rightarrow key(E) \text{ does not hold}$$

The constraints:

$$key(E'_{new}) = X$$

The corresponding predicate $\mathcal{A}$:

$$\forall x_1, x.(E_{new}(x, x_1) = 1 \Leftrightarrow \exists a.E(x, a, x_1) = 1) \wedge \\ \forall a, x.(E'_{new}(x, a) = 1 \Leftrightarrow \exists x_1.E(x, a, x_1) = 1)$$

The corresponding computable queries f and g:

$$E_{new} := \{(x, x_1) \mid \exists a.(x, a, x_1) \in E\} \| \\ E'_{new} := \{(x, a) \mid \exists x_1.(x, a, x_1) \in E\}$$

and

$$E := \{(x, a, x_1) \mid (x, a) \in E'_{new} \wedge (x, x_1) \in E_{new}\}$$

respectively.

Example 6.24. *An example is shown in Figure 6.17.*

□

***Rule* 26.** For a path $P \equiv R_1 - R - R_2$ with $comp(R) = \{r_1 : R_1, r_2 : R_2\}$ and $attr(R) = \emptyset$ such that the path functional dependency $X \rightarrow key(R_2)$ holds for a generalised subset X of $attr(R_1)$ replace $key(R)$ by $\{r_1 : R_1\}$.

$$\frac{\begin{array}{ll}\mathfrak{M} \triangleright & R_1 = r_1 : id \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times t_2 \rightarrow \{\mathbb{1}\} \\ & R = K : (r_1 : ref \times r_2 : ref) \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \triangleright & R_1 = r_1 : id \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times t_2 \rightarrow \{\mathbb{1}\} \\ & R = K : (r_1 : ref) \times r_2 : ref \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

where K represents the key.
The side condition φ:

$$X \rightarrow key(R_2)$$

Since the change has no impact on the capacity of the information in either machines, no corresponding predicate $\mathcal{A}$ or computable query f or g is required.

Example 6.25. *Let us assume the following:*

$R_1 = \{(1, a), (2, a)\},\ R = \{(1, 1, 2), (2, 2, 2)\},\ R_2 = \{(1, b), (2, c)\}$

where the 1st column in the tuples are the keys, the 2nd and 3rd in R are the references to R_1 and R_2 respectively. This is an example with the conditions set in the Rule 26 held. We can apply the rule to obtain:

$R = \{(1, 2), (2, 2)\}$

with the original key removed and the reference to R_1 as the key instead.

□

***Rule* 27.** For a path $P \equiv R_1 - \cdots - R_n$ such that the path functional dependency $X \rightarrow A$ holds for a generalised subset X of $attr(R_1)$ and $A \in attr(R_n)$ add a new entity type E_{new} with $attr(E_{new}) = X \cup \{A\}$ and $key(E_{new}) = X$.

$$\frac{\begin{array}{ll}\mathfrak{M} \triangleright & R_1 = r_1 : id \times [r_2 : ref] \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \rightarrow \{\mathbb{1}\} \\ & \dots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times A : t_a \times t_n \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\mathfrak{M}^* \triangleright & R_1 = r_1 : id \times [r_2 : ref] \times X : t_x \times t_1 \rightarrow \{\mathbb{1}\} \\ & R_2 = r_2 : id \times [r_1 : ref] \times [r_3 : ref] \times t_1 \rightarrow \{\mathbb{1}\} \\ & \dots \\ & R_n = r_n : id \times [r_{n-1} : ref] \times A : t_a \times t_n \rightarrow \{\mathbb{1}\} \\ & E_{new} = X : t_x \times A : t_a \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

where $[r_i : ref](1 \leq i \leq n)$ models a possible reference to type R_i, due to the path definition $P \equiv R_1 - \cdots - R_n$ covering all possible path directions, as depicted in Figure

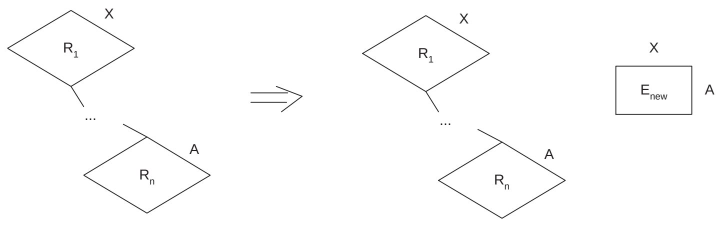

Figure 6.18: The relationship types before and after application of Rule 27

6.4. We say that $[r_i : ref]$ is realised, denoted by $[r_{i+1} : ref] \equiv r_{i+1} : ref$ iff $[r_i : ref]$ models an actual reference to R_i.

The side condition φ:

$$X \to A \text{ on } P \equiv R_1 - \cdots - R_n$$

The constraint:

$$key(E_{new}) = X$$

The corresponding predicate $\mathcal{A}$:

$$\begin{gathered} \forall x, a.(E_{new}(x,a) = 1 \Leftrightarrow \\ \exists r_1, \ldots r_n, x_1, \ldots, x_n.(R_1(\pi_{T_1}(r_1, r_2, x, x_1)) = 1 \wedge \\ R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, a, x_n)) = 1) \end{gathered}$$

where T_i is the source type of R_i, i.e. we have $R_i = T_i \to \{\mathbb{1}\}$. $\pi_{T_i}(r_i, r_{i-1}, r_{i+1}, x_i)$ projects the tuple $(r_i, r_{i-1}, r_{i+1}, x_i)$ onto T_i.

The corresponding computable queries f:

$$\begin{aligned} E_{new} := \{(x,a) \mid \exists r_1, \ldots r_n, x_1, \ldots, x_n.(R_1(\pi_{T_1}(r_1, r_2, x, x_1)) = 1 \wedge \\ R_2(\pi_{T_2}(r_2, r_1, r_3, x_2)) = 1 \wedge \cdots \wedge R_n(\pi_{T_n}(r_n, r_{n-1}, a, x_n)) = 1)\} \end{aligned}$$

and g an identity function.

Example 6.26. *An example is shown in Figure 6.18.*

□

Rule 28. Let $X \cup Y \cup Z = attr(R) \cup comp(R)$ such that $R[X,Y] \bowtie R[X,Z] \subseteq R$ holds, then replace R by $R_{1,new}$ and $R_{2,new}$ with $attr(R_{1,new}) = X \cup Y$ and $attr(R_{2,new}) = X \cup Z$.

$$\frac{\mathfrak{M} \rhd \quad R = X : t_x \times Y : t_y \times Z : t_z \to \{\mathbb{1}\}}{\begin{aligned} \mathfrak{M}^* \rhd \quad & R_{1,new} = X : t_x \times Y : t_y \to \{\mathbb{1}\} \\ & R_{2,new} = X : t_x \times Z : t_z \to \{\mathbb{1}\} \end{aligned}} \; \varphi$$

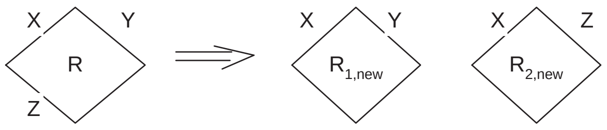

Figure 6.19: The relationship types before and after application of Rule 28

with side condition φ:

$$R[X,Y] \bowtie R[X,Z] \subseteq R$$

The corresponding predicate $\mathcal{A}$:

$$\forall x,y,z.((R(x,y,z) = 1 \Leftrightarrow R_{1,new}(x,y) = 1 \wedge R_{2,new}(x,z) = 1)$$

The corresponding computable queries f and g:

$$R_{1,new} := \{(x,y) \mid \exists z.R(x,y,z) = 1\}\|$$
$$R_{2,new} := \{(x,z) \mid \exists y.R(x,y,z) = 1\}$$

and

$$R := \{(x,y,z) \mid (x,y) \in R_{1,new} \wedge (x,z) \in R_{2,new}\}$$

respectively.

Example 6.27. *An example is shown in Figure 6.19.*

□

6.6 Dialogue Types

We have seen that the major functionality on the OLAP tier can be expressed by views that are extended by operations, such as roll-up or drill-down, that is exactly the idea underlying dialogue types. However, we simplify the original definition from [99] omitting the subtle distinction between hidden and visible parts. We also omit hierarchies of dialogue types.

Definition 6.10. A *dialogue type* D over a HERM schema $(\mathcal{S},\Sigma)$ consists of a view $V_D = (\mathcal{S}_D, q_D)$, in which $\mathcal{S}_D$ consists of a single entity type E, and a set $\mathcal{O}$ of *dialogue operations.* Each dialogue operation (*d-operation* for short) in $\mathcal{O}$ consists of

- an operation name *op*,
- a list of input parameters $i_1 : D_1, \dots, i_k : D_k$ with domain names D_i,
- an (optional) output domain D_{out},

- a subattribute *sel* of the representing attribute X_E, and
- a d-operation body, which is built from usual programming constructs operating on instances over $(\mathcal{S}, \Sigma)$ and constructs for creating and deleting dialogue objects.

Whenever we are given an instance *db* over $(\mathcal{S}, \Sigma)$, the defining query q_D produces an instance over $\mathcal{S}_D$, i.e. a set of pairs (i, v) with $i \in ID$ and $v \in dom(X_E)$, each of which will be called a *dialogue object of type D*. At any time only a subset of $q_D(db)$ will be available, the *set of active dialogue objects*. These represent the active user dialogues in an abstract way.

The presented value v may be projected to $\pi_{sel}^{X_E}(v)$, which represents the data that must be selected by the user as a prerequisite for executing operation *op*. Once this data is selected and the operation *op* is started, further input for the parameters $i_1, \dots, i_k$ will be requested from the user – using e.g. so-called dialogue boxes [99] – and the execution of *op* will update the database *db* and result in a new set of active dialogue objects.

6.7 Transformation Rules for Dialogue Types

As user dialogues are an invaluable source of information in requirements engineering, we may usually assume that we know about dialogue objects before the defining queries and the underlying database schema is fixed. Therefore, view integration is an unavoidable design task. In addition, we will always be confronted with the desire to rearrange the “data marts”, i.e. the dialogue types that define the OLAP functionality. This problem also appears in database applications other than OLAP.

Therefore, the additional problem is to adapt the d-operations that are used for the functionality presented to a data warehouse user. For this we define further transformation rules. However, these additional transformation rules have to be understood as follow-on rules for the case that one of the rules Rule 1-28 is not just applied to schemata or views, but to dialogue types. Thus, we obtain additional changes to the selection attribute *sel* and the body of d-operations. As these changes are the corresponding changes resulted from view integration in dialogue types, they are so simple and straight forward that we will not provide formal rules as that given above.

Rule 29. In case Rule 1 is applied to a dialogue type, whenever $X(A'_1, \dots, A'_k)$ $(k \leq m)$ appears in *sel* or in the body of a d-operation, replace it by $A'_1, \dots, A'_k$.

Rule 30. In case Rule 2 is applied to a dialogue type, whenever r appears in *sel* or in the body of a d-operation, replace it by $r_1^{(r)}, \dots, r_n^{(r)}$.

We may ignore Rule 4 and 5, as clusters have not been allowed in dialogue types. There is also nothing to add for Rule 6, as this introduces a new type, so operations have to be defined for that type.

Rule 31. In case Rule 7 is applied to a dialogue type omit A in *sel* or in the body of a d-operation, whenever it appears.

Rule 32. In case Rule 8 is applied to a dialogue type omit $r_1, \dots, r_n$ in *sel* or in the body of a d-operation, whenever it appears.

These extensions capture the first group of rules dealing with schema restructuring. For the second group of rules dealing with the shifting of attributes we obtain the following extension rules in case the rules are applied to dialogue types.

Rule 33. In case Rule 9 is applied to a dialogue type omit A_i in *sel* and the body of operations associated with $R_{i,new}$, whenever it appears in *sel* or the body of an operation associated with R_i.

Rule 34. In case Rule 10 is applied to a dialogue type omit A in *sel* and the body of operations associated with R_{new}, whenever it appears in *sel* or the body of an operation associated with R.

Note that the last two extension rules have no effect on R_{new} or $R_{i,new}$, as the extension of the selection attribute or the body of an operation has to be defined for these new types.

Rule 35. In case Rule 11 is applied to a dialogue type replace r_n by $r_{1,new}$ in *sel* and the body of operations associated with R_{new}.

Rule 36. In case Rule 12 is applied to a dialogue type omit A in *sel* and the body of operations associated with $R_{n,new}$.

For the third group of rules, i.e. Rule 13-17 dealing with schema extension we cannot define reasonable extension rules for dialogue types, as we always have to deal with completely new types. The same applies to Rule 18-20, i.e. the group of rules dealing with type integration.

Finally, for the group of rules dealing with integrity constraints only Rules 22, 24 and 25 give rise to the following three extension rules for dialogue types.

Rule 37. In case Rule 22 is applied to a dialogue type replace r_2 by r_{new} in *sel* and the body of operations, whenever it appears.

Rule 38. In case Rule 24 is applied to a dialogue type replace r_2 by $r_{2,new}$ in *sel* and the body of operations, whenever it appears.

Rule 39. In case Rule 25 is applied to a dialogue type remove A in *sel* and the body of operations, whenever it appears.

Chapter 7

Case Studies

In this chapter, we present four case studies in data warehouse and OLAP systems development. First, we show how a data warehouse is built incrementally from data marts. The second case deals with dynamic data warehouse design with focus on performance issues. In the third case we address the issue of distribution design, for which we propose to analyse and optimise query and maintenance costs. Finally we look at a case with specific application in OLAP for supporting management decision making, namely the linear regression and time series analysis. Our purpose for the case studies is to show how we deal with the common issues in data warehouse and OLAP systems design using the refinement-based design method.

7.1 Adding a New Data Mart

As we discussed in the early chapters, our method allows to build a data warehouse from data marts while preserving system integrity. This is achieved by applying the schema integration technique introduced in Chapter 6. In the following, we show step by step how a new data mart is added to the existing system, using our refinement-based design method following the integration process and by applying the schema transformation rules. This design step is classified under the requirement capturing phase of our refinement-based design method.

The data warehouse design process is complex due to its involvement with many business process areas, each of which are with many design concerns, such as schemata, functionalities and performance, etc. However, our discussion of adding a new data mart in the following will not show how to build an optimal data mart, but focus on the issue of data integrity by schema integration.

7.1.1 Example: CRM for Grocery Store

We take CRM (Customer Relationship Management) as an example data mart to add into our grocery store data warehouse, in particular, using the ground model in TASM given in Section 5.1.5. Designing data warehouses for CRM is discussed in many publications, e.g. [25, 110, 58]. As shown in [25], it is complex to design the schema for CRM. In their case study they obtained a complicated data warehouse schemata with 3 facts and 19 dimensions in a starter model. However, the principle of our ASM-based method is to simplify it by abstraction and stepwise refinements. For example, the basic requirements, which are modelled by a set of analysis types in [25], are basically a set of new OLAP views, namely a set of user reports/queries. Furthermore, our experience shows that

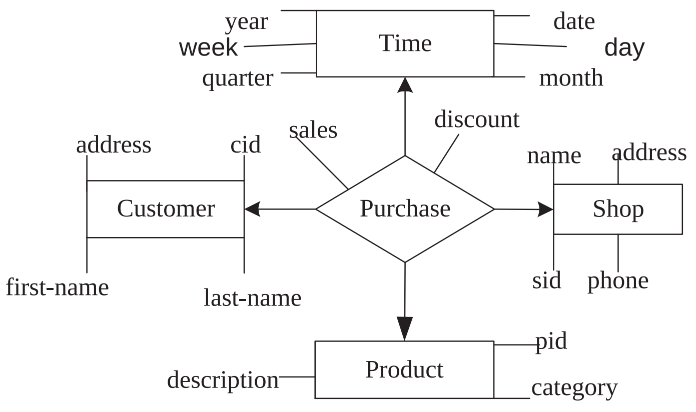

Figure 7.1: The data warehouse schema for CRM

the requirements from users usually come in form of reports or queries. Therefore, as more reports/queries can be added in a similar fashion later on, we may take a report on customer purchase by customer (customer id, last name, first name), shop, product, month, quarter and year with details, such as total sales and total discount, to support customer profitability analysis as a start. Based on this requirement, we derive a data schema as shown in Figure 7.1. This will then be integrated with the data warehouse schema from Chapter 5.

7.1.2 Incorporate CRM

We use the refinement-based method introduced in Chapter 5 to incorporate the new OLAP view into the existing ground model, which is constructed in section 5.1, by applying the following steps:

1. *Add a new rule to the OLAP ASM*: we add a new rule for customer purchase report. This rule will be exported for DW ASM to use.

```
$create\_V\_customer\_purchase$ = forall $c, ln, fn, s, p, m, q, y$ with
    $\exists n, a, ph.(s, n, a, ph) \in \text{Shop\_CRM} \wedge$
    $\exists de, ca.(p, de, ca) \in \text{Product} \wedge$
    $\exists d, d', w.(d, d', w, m, q, y) \in \text{Time} \wedge$
    $\exists a'.(c, ln, fn, a') \in \text{Customer\_DW}$
    do let $S = src[0, \pi_{s'}, +]$
            $(\{(c, s, p, d, s') \mid \exists dic.$
            $(i, s, p, d, s', dic) \in \text{Purchase\_CRM} \wedge$
            $d.month = m \wedge d.year = y\})$
        $D = src[0, \pi_{dic}, +]$
            $(\{(i, s, p, d, dic) \mid \exists s'.$
            $(i, s, p, d, s', dic) \in \text{Purchase\_CRM} \wedge$
            $d.month = m \wedge d.year = y\})$
        in $\text{V\_customer\_purchase}(c, ln, fn, s, p, m, q, y, S, D) := 1$
    enddo
```

2. Add a new controlled function to the OLAP ASM: apply the schema transformation

rule 13, to add the OLAP view V_customer_purchase and the corresponding data mart view DM_V_customer_purchase. These two functions will be exported and imported for DW ASM to use.

$$\begin{aligned}&\text{OLAP-ASM}^* \rhd \text{V\_customer\_purchase}\\ &\quad = cid \times last\_name \times first\_name \times shop \times product\\ &\qquad \times month \times qtr \times year \times sales \times discount \rightarrow \{\mathbb{1}\}\end{aligned}$$

$$\begin{aligned}&\text{OLAP-ASM}^* \rhd \text{DM-V\_customer\_purchase}\\ &\quad = dm \times cid \times last\_name \times first\_name \times shop \times product\\ &\qquad \times month \times qtr \times year \times sales \times discount \rightarrow \{\mathbb{1}\}\end{aligned}$$

3. *Add new controlled function(s) to the DW ASM*: apply the schema transformation rule 13, to add the following functions to DW ASM, as the current data model does not support them. We will export the schema Customer_DW, and import it in DB ASM. Note that time and product do not need to be added since they are identical to the existing relations.

$$\text{DW-ASM}^* \rhd Address =_t address : town \times region \times state$$

$$\begin{aligned}&\text{DW-ASM}^* \rhd \text{Customer\_DW}\\ &\quad = cid \times last\_name \times first\_name \times Address \rightarrow \{\mathbb{1}\}\end{aligned}$$

$$\begin{aligned}&\text{DW-ASM}^* \rhd \text{shop\_CRM}\\ &\quad = sid \times name \times Address \times phone \rightarrow \{\mathbb{1}\}\end{aligned}$$

$$\begin{aligned}&\text{DW-ASM}^* \rhd \text{Purchase\_CRM}\\ &\quad = cid \times sid \times pis \times date \times sales \times discount \rightarrow \{\mathbb{1}\}\end{aligned}$$

4. *Integrate controlled functions on the DW ASM*: as we have added two new functions in step 3, we carry out the integration process, i.e. the 6-step process, as described in Section 6.4.

   (a) Homogenisation of the schemata: in this step, we apply Rule 1 to break the address of the shop into town, region and state as follows:

$$\frac{\text{DW-ASM} \rhd \text{Shop\_CRM} = sid \times name \times Address \times phone \rightarrow \{\mathbb{1}\}}{\begin{aligned}\text{DW-ASM}^* \rhd \;\; &\text{Shop\_CRM}_{new} = sid \times name \times town\\ &\times region \times state \times phone \rightarrow \{\mathbb{1}\}\end{aligned}}$$

   (b) Adding inter-schema integrity constraints: not required for our example.

   (c) Modulisation: not required in our example.

(d) Type integration: in this step, we apply Rule 18 to integrate Shop with Shop_CRM$_{new}$, Purchase with Purchase_CRM, as follows:

$$\frac{\begin{array}{ll}\text{DW-ASM} \rhd & \text{Shop} = \\ & sid \times name \times town \times region \times state \times phone \to \{\mathbb{1}\} \\ & \text{Shop\_CRM}_{new} = \\ & sid \times name \times town \times region \times state \times phone \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\text{DW-ASM}^* \rhd & \text{Shop}_{new} = \\ & sid \times name \times town \times region \times state \times phone \to \{\mathbb{1}\}\end{array}}\ \varphi$$

which is a special case of type integration for two identical schemas, and

$$\frac{\begin{array}{ll}\text{DW-ASM} \rhd & \text{Purchase} = \\ & sid \times cid \times pid \times date \times \\ & qty \times sales \times profit \to \{\mathbb{1}\} \\ & \text{Purchase\_CRM} = \\ & sid \times cid \times pid \times date \times \\ & sales \times discount \to \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\text{DW-ASM}^* \rhd & \text{Purchase}_{new} = \\ & sid \times cid \times pid \times date \times \\ & qty \times sales \times profit \times discount \to \{\mathbb{1}\}\end{array}}\ \varphi$$

The side condition φ:

$$K_1 = K_2 \wedge R_1[K_1 \cup X] = h(R_2[K_2 \cup Y]),$$

is satisfied, with the keys in both schemas being $sid \times cid \times pid \times date$, and h the identity function.

(e) Path integration: not required in our example.

(f) Other integrity constraints: not required in our example.

Instead of exporting and importing the schemata with the new names, we can rename the schemata Shop$_{new}$ back to Shop and Purchase$_{new}$ back to Purchase, so we avoid the changes in export and import. The new schema after the integration is shown as Figure 7.2.

5. *Add controlled functions to the DB ASM*: we combine this step with the schema integration to extend the schema Buys to track the discount. This can be done by applying Rule 14 as follows:

$$\frac{\text{DB-ASM} \rhd \text{Buys} = time \times cid \times sid \times pid \times quantity \to \{\mathbb{1}\}}{\text{DB-ASM}^* \rhd \text{Buys}_{new} = time \times cid \times sid \times pid \times quantity \times discount \to \{\mathbb{1}\}}$$

Again we will rename Buys$_{new}$ back to Buys to avoid unnecessary changes. The new schema for the operational DB is shown in Figure 7.3.

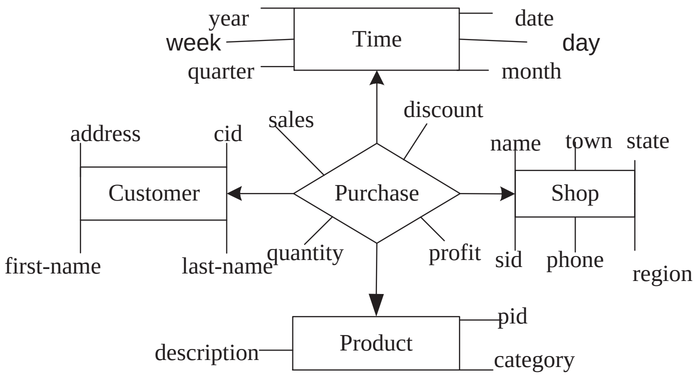

Figure 7.2: The data warehouse schema after incorporating CRM

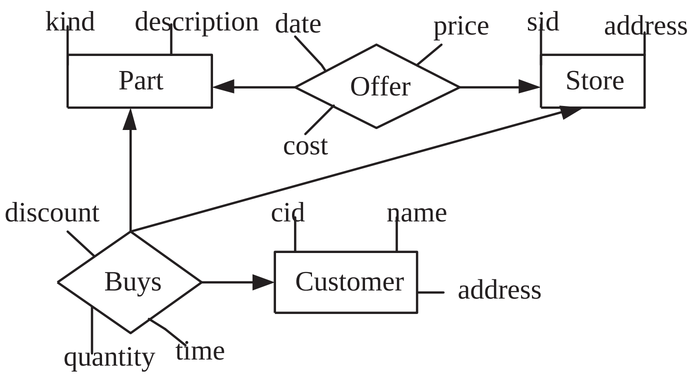

Figure 7.3: The operational DB schema after incorporating CRM

6. *Change the rules on DW ASM*: in this step we propagate changes into the impacted rules, e.g. *extract_purchase*, a new rule for Customer_DW, and *open_datamart*(dm) as follows:

```
$extract\_purchase$ = forall $i,\ p,\ s,\ d,\ p',\ c$ with
 $\exists t.(i,p,s,t,p',c,d) \in \pi_{cid,pid,sid,time,price,cost,discount}$
   $(\text{Buys} \bowtie \text{Customer\_DB} \bowtie \text{Part} \bowtie \text{Store} \bowtie \text{Offer}) \wedge t.date = d$
   do let $Q = src[0, \pi_q, +](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge t.date = d\})$
     $D = src[0, \pi_d, +](\{(t,d) \mid (i,s,p,t,d) \in Buys \wedge$
       $t.date = d\})$ $S = Q * p' - D$ $P = Q * (p' - c) - D$
    in $\text{Purchase}(i,p,s,d,Q,S,P,D) := 1$ enddo

$extract\_customer$ = forall $i,n,d,a$ with
 $(i,n,d,a) \in \text{Customer\_DB}$
 do $\text{Customer\_DW}(i, n.first, n.last, a) := 1$ enddo

$open\_datamart(dm)$ = case the-matching-view$(dm)$ of
 V_sales : $create\_V\_sales$;
   forall $s,r,st,m,q,y,S$ with
     $(s,r,st,m,q,y,S) \in$ V_sales do
       DM-V_sales$(dm,s,r,st,m,q,y,S) := 1$ enddo
 V_customer_purchase : $create\_V\_customer\_purchase$;
   forall $c,ln,fn,s,p,m,q,y,S,D$ with
     $(c,ln,fn,s,p,m,q,y,S,D) \in$ V_customer_purchase do
       DM-V_customer_purchase$(dm,c,ln,fn,s,p,m,q,y,S,D) := 1$ enddo
 endcase

$close\_datamart(dm)$ = case the-matching-view$(dm)$ of
 V_sales : forall $s,r,st,m,q,y,S$ with
           (the-datamart$(op),s,r,st,m,q,y,S) \in$ DM-V_sales do
             DM-V_sales(the-datamart$(op),s,r,st,m,q,y,S) := \perp$ enddo
 enddo endcase
```

We export the rule *extract_customer* and import it in DB ASM.

7. *Change the rules on DB ASM*: we change the main rule to include data extraction for Customer_DW as follows:

```
$main$ = if r-type(req) = $extract$ then
 $extract\_purchase$ || $extract\_shop$ || $extract\_product$
 $extract\_time$ || $extract\_customer$
endif
```

8. *Change the functions/rules on OLAP ASM*: due to the newly added OLAP view and the renaming of schemata, the rules *close_datamart* and *create_V_customer_purchase* will be changed as follows:

```
$close\_datamart(dm)$ = case the-matching-view$(dm)$ of
 V_sales : forall $s,r,st,m,q,y,S$ with
   (the-datamart$(op),s,r,st,m,q,y,S) \in$ DM-V_sales do
```

```
        DM-V_sales(the-datamart($op$), $s, r, st, m, q, y, S$) := $\perp$ enddo
    V_customer_purchase : forall $c, ln, fn, s, p, m, q, y, S, D$ with
      (the-datamart($op$), $c, ln, fn, s, p, m, q, y, S, D$) $\in$ DM-V_customer_purchase do
      DM-V_customer_purchase(the-datamart($op$), $c, ln, fn, s, p, m, q, y, S, D$) := $\perp$
    enddo endcase
```

```
  $create\_V\_customer\_purchase$ = forall $c, ln, fn, s, p, m, q, y$ with
      $\exists n, t, r, st, ph.$Shop$(s, n, t, r, st, ph) = 1 \wedge$
      $\exists de, ca.(p, de, ca) \in$ Product$\wedge$
      $\exists d, d', w.(d, d', w, m, q, y) \in$ Time$\wedge$
      $\exists a'.(c, ln, fn, a') \in$ Customer_DW
      do let $S = src[0, \pi_{s'}, +]$
            $(\{(c, s, p, d, s') \mid \exists q', p', dic.$
            $(i, s, p, d, q', s', p', dic) \in$ Purchase $\wedge$
            $d.month = m \wedge d.year = y\})$
         $D = src[0, \pi_{dic}, +]$
            $(\{(i, s, p, d, dic) \mid \exists q', p', s'.$
            $(i, s, p, d, q', s', p', dic) \in$ Purchase $\wedge$
            $d.month = m \wedge d.year = y\})$
      in V_customer_purchase$(c, ln, fn, s, p, m, q, y, S, D) := 1$
    enddo
```

The three refined ASM models after the CRM is incorporated are given as follows:

```
TASM DB-ASM
IMPORT
  DW-ASM(Shop, Product, Time, Purchase, Customer_DW,
  $extract\_purchase$, $extract\_shop$,
  $extract\_product$, $extract\_time$, $extract\_customer$)
EXPORT
  Store, Part, Buys, Offer
SIGNATURE
  Store:$sid \times name \times size \times address \rightarrow \{\mathbb{1}\}$,
  Part:$pid \times kind \times description \rightarrow \{\mathbb{1}\}$,
  Customer_DB:$cid \times name \times dob \times address \rightarrow \{\mathbb{1}\}$,
  Buys:$time \times cid \times sid \times pid \times quantity \times discount \rightarrow \{\mathbb{1}\}$,
  Offer:$pid \times sid \times date \times price \times cost \rightarrow \{\mathbb{1}\}$,
  r-type (external), req (external)
BODY
  $main$ = if r-type(req) = $extract$ then
    $extract\_purchase$ || $extract\_shop$
    $extract\_product$ || $extract\_time$ || $extract\_customer$
  endif
```

```
TASM DW-ASM
IMPORT
  DB-ASM(Store, Part, Customer_DB, Buys, Offer)),
  OLAP-ASM(V_sales, DM-V_sales, V_customer_purchase,
    DM-V_customer_purchase, the-datamart, the-matching-view,
    $create\_V\_sales$, $create\_V\_customer\_purchase$ )
EXPORT
  Shop, Product, Time, Purchase, Customer_DW
  $extract\_purchase$, $extract\_shop$,
  $extract\_product$, $extract\_time$, $extract\_customer$
SIGNATURE
  Shop:$sid \times name \times town \times region \times state \times phone \rightarrow \{\mathbb{1}\}$,
  Product:$pid \times category \times description \rightarrow \{\mathbb{1}\}$,
  Time:$date \times day \times week \times month \times quarter \times year \rightarrow \{\mathbb{1}\}$,
  Customer_DW: $cid \times last\_name \times first\_name \times address \rightarrow \{\mathbb{1}\}$
  Purchase:$cid \times sid \times pid \times date \times qty \times sales \times profit \rightarrow \{\mathbb{1}\}$,
  r-type (external), req (external)
BODY
  $main$ =if r-type(req)=$open\text{-}datamart$ then
    $open\_datamart(the\text{-}datamart(req))$ endif

  $extract\_purchase$ = forall $i$, $p$, $s$, $d$, $p'$, $c$ with
    $\exists t.(i,p,s,t,p',c,d) \in \pi_{cid,pid,sid,time,price,cost,discount}$
        (Buys $\bowtie$ Customer_DB $\bowtie$ Part $\bowtie$ Store $\bowtie$ Offer) $\wedge\ t.date = d$
        do let $Q = src[0,\pi_q,+](\{(t,q) \mid (i,s,p,t,q) \in Buys \wedge t.date = d\})$
            $D = src[0,\pi_d,+](\{(t,d) \mid (i,s,p,t,d) \in Buys \wedge$
                $t.date = d\})$ $S = Q * p' - D$ $P = Q * (p' - c) - D$
         in Purchase$(i,p,s,d,Q,S,P,D)$ := 1 enddo

  $extract\_shop$ = forall $s,n,a$ with
    $\exists s'.(s,n,s',a) \in$ Store
    do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
         in Shop$(s,n,t,r,st,ph)$ := 1 enddo

  $extract\_product$ = forall $p,k,d$ with
    $(p,k,d) \in$ Part
    do let $p' = p, c = k, d' = d$
         in Product$(p',c,d')$ := 1 enddo

  $extract\_time$ = forall $t$ with
    $\exists\ c,p,s,q.(c,p,s,q,t) \in$ Buys
    do if Time$(t.date, t.day, t.week, t.quarter, t.month, t.year) = \perp$
         then Time$(t.date, t.day, t.week, t.quarter, t.month, t.year)$ := 1
    enddo

  $extract\_customer$ = forall $i,n,d,a$ with
    $(i,n,d,a) \in$ Customer_DB
    do let $i' = i$ $fn = n.first$ $ln = n.last$ $a' = a$
         in Customer_DW$(i',fn,ln,a')$ := 1 enddo
```

```
$open\_datamart(dm)$ = case the-matching-view($dm$) of
 V_sales : $create\_V\_sales$;
    forall $s, r, st, m, q, y, S$ with
      $(s, r, st, m, q, y, S) \in$ V_sales do
        DM-V_sales($dm, s, r, st, m, q, y, S$) := 1 enddo
 V_customer_purchase : $create\_V\_customer\_purchase$;
    forall $c, ln, fn, s, p, m, q, y, S, D$ with
      $(c, ln, fn, s, p, m, q, y, S, D) \in$ V_customer_purchase do
        DM-V_customer_purchase($dm, c, ln, fn, s, p, m, q, y, S, D$) := 1 enddo
 endcase
```

```
TASM OLAP-ASM
IMPORT DW-ASM(Shop, Product, Time, Purchase)
EXPORT V_sales, DM-V_sales, $create\_V\_sales$,
       V_customer_purchase, DM-V_customer_purchase, $create\_V\_customer\_purchase$,
       the-datamart, the-matching-view
SIGNATURE
  V_sales:$sid \times region \times state \times month \times quarter \times year \times$
       $sales \times profit \rightarrow \{\mathbb{1}\}$,
  DM-V_sales:$dm \times sid \times region \times state \times month \times quarter \times$
       $year \times sales \rightarrow \{\mathbb{1}\}$ ,
  V_customer_purchase:$cid \times last\_name \times first\_name \times shop \times product$
       $\times month \times qtr \times year \times sales \times discount \rightarrow \{\mathbb{1}\}$
  DM-V_customer_purchase:$dm \times cid \times last\_name \times first\_name \times shop \times product$
       $\times month \times qtr \times year \times sales \times discount \rightarrow \{\mathbb{1}\}$
  o-type: $op \rightarrow \{open, close, quit\}$,
  owner: $datamart \rightarrow user$,
  issuer: $op \rightarrow user$,
  the-datamart: $op \rightarrow datamart$,
  the-view: $op \rightarrow view$,
  the-matching-view: $datamart \rightarrow view$,
  op: $operation$ (external)
BODY
  $main$ = if o-type(op) = $login$ then $LOGIN$
     elsif if registered(issuer(op))=1 then
        if o-type(op) = $open$ then $OPEN$
     elsif o-type(op) = $close$ then $CLOSE$
     elsif o-type(op) = $quit$ then $QUIT$
  endif

  $LOGIN$ =
     registered(issuer(op)):=1

  $OPEN$ =
     import $dm$
        datamart($dm$) := 1
        owner($dm$) := issuer($op$)
        the-matching-view($dm$) := the-view($op$)
        import $reqst$
            let $reqst = (open\text{-}datamart, \mathrm{dm})$ in
            $\mathrm{request}(reqst) := 1$
        end-import;
    end-import;


$CLOSE$ =
    $close\_datamart(\text{the-datamart}(op))$;
    $\text{owner}(\text{the-datamart}(op)) := \perp$
    $\text{datamart}(\text{the-datamart}(op)) := \perp$


$QUIT$ =
    let $usr = issuer(op)$ in
        forall $dm$ with $\text{owner}(dm) = usr$
            do $close\_datamart(dm)$
                $\text{datamart}(dm) := \perp$ $\text{owner}(dm) := \perp$ enddo
        $\text{registered}(usr) := \perp$


$close\_datamart(dm)$ = case $\text{the-matching-view}(dm)$ of
  $\text{V\_sales}$ : forall $s, r, st, m, q, y, S$ with
    $(\text{the-datamart}(op), s, r, st, m, q, y, S) \in \text{DM-V\_sales}$ do
        $\text{DM-V\_sales}(\text{the-datamart}(op), s, r, st, m, q, y, S) := \perp$ enddo
  $\text{V\_customer\_purchase}$ : forall $c, ln, fn, s, p, m, q, y, S, D$ with
    $(\text{the-datamart}(op), c, ln, fn, s, p, m, q, y, S, D) \in$
    $\text{DM-V\_customer\_purchase}$ do
    $\text{DM-V\_customer\_purchase}(\text{the-datamart}(op),$
                                    $c, ln, fn, s, p, m, q, y, S, D) := \perp$
  enddo endcase


$create\_V\_sales$ = forall $s, r, st, m, q, y$ with
    $\exists n, t, ph.(s, n, t, r, st, ph) \in \text{Shop} \wedge$
    $\exists d, d', w.(d, d', w, m, q, y) \in \text{Time}$
    do let $S = src[0, \pi_{s'}, +]$
        $(\{(i, s, p, d, s') \mid \exists q', p'.$
        $(i, s, p, d, q', s', p') \in \text{Purchase} \wedge$
        $d.month = m \wedge d.year = y\})$
    in $\text{V\_sales}(s, r, st, m, q, y, S) := 1$
  enddo


$create\_V\_customer\_purchase$ = forall $c, ln, fn, s, p, m, q, y$ with
    $\exists n, t, r, st, ph.\text{Shop}(s, n, t, r, st, ph) = 1 \wedge$
    $\exists de, ca.(p, de, ca) \in \text{Product} \wedge$
    $\exists d, d', w.(d, d', w, m, q, y) \in \text{Time} \wedge$
    $\exists a'.(c, ln, fn, a') \in \text{Customer\_DW}$
    do let $S = src[0, \pi_{s'}, +]$
        $(\{(c, s, p, d, s') \mid \exists q', p', dic.$
        $(i, s, p, d, q', s', p', dic) \in \text{Purchase} \wedge$
```

```
            $d.month = m \wedge d.year = y\})$
        $D = src[0, \pi_{dic}, +]$
            $(\{(i, s, p, d, dic) \mid \exists q', p', s'.$
            $(i, s, p, d, q', s', p', dic) \in \text{Purchase} \wedge$
            $d.month = m \wedge d.year = y\})$
      in $\text{V\_customer\_purchase}(c, ln, fn, s, p, m, q, y, S, D) := 1$
  enddo
```

7.2 Dynamic Data Warehouse Design

As the requirements in business analysis change over time, data warehouse design becomes an ongoing task. Theodoratos et al approach this issue as dynamic data warehouse design in [109, 108]. They aim at preserving the set of existing materialised views and extending it when needed in rewriting the new OLAP queries. The drawback of their approach is that it may compromise the optimal query performance and view maintenance due to the constraint. In our work, we suggest materialising a new view under a space constraint if it is beneficial for computing other OLAP queries, even if it can be computed from the materialised views, and for the benefit of view maintenance to apply view integration techniques when a new view is added.

In the following, we present first the cost model for query evaluation and view maintenance and the benefit models for determining view selection, and then the view selection process. Finally we present some cases for view selection and dynamic data warehouse design.

7.2.1 Cost and benefit Model

For simplicity, we adopt the basic idea from [48] in estimation of the query evaluation cost and view maintenance cost. That is, we use the size of a view v as its maintenance cost and the query evaluation cost of a query q, if q is computed totally from v:

$$qcost(q, v) = s(v)$$

Similarly, we have the view maintenance cost:

$$mcost(v) = s(v)$$

Further we introduce a notion of benefit for comparing two materialised views v_1 over v_2 in computing query q of frequency f as:

$$b(v_1, v_2) = (s(v_2) - s(v_1)) \times (f + 1)$$

where both the query cost and the view maintenance cost are considered. If b is positive, it means materialising v_1 is more beneficial.

In the case that one of the views is already materialised, thus maintaining it does not involve additional cost, the benefit of using existing materialized view v_1 to compute query q over creating a new view v for query q is:

$$bx(v_1) = (s(v) - s(v_1)) \times f + s(v)$$

For deciding if materialising a new view v is more favorable for another OLAP view o, the benefit is estimated:

$$bz(o) = ((s(v(o)) - s(v)) \times f(o)$$

where $v(o)$ is a materialised view which is used for computing o. If the sum over all positive $bz(o)$ is greater than $s(v)$, we consider it is beneficial to materialise v and therefore we rewrite the corresponding queries using v.

The above estimation is rather simple. However, a more comprehensive estimation can be applied easily. For more involved models see e.g. [109, 41]. As how to best estimate the costs is not our focus, we adopt this simplistic but still reasonable approach.

7.2.2 View Selection Process

The basic idea of our view selection is that under the given constraint, S, the space constraint, we always materialise a new view if it cannot be computed from the existing materialised views, or it is more beneficial for computing other OLAP queries. Although we should be more concerned with the time for refreshing the materialised views than the storage space, our justification is that the larger the total size of the views, the longer it will take to maintain. Our selection process is invoked whenever there is a new OLAP view added, we invoke the selection process to check if the set of existing materialised views can be used to compute the OLAP view. In order to do so, we define a notion of fineness to compare two views, such that we can compute the less fine one from the finer one. We agree with [61] that there is not much to gain by computing a view using multiple views when we are considering typical OLAP queries involving aggregations.

Definition 7.1. A view(query) v_1 is called to be *finer* than v_2, denoted as $v_1 \succ v_2$, if v_2 is computable from v_1 by aggregating operations.

Example 7.1. If v_1 is the view of *sales by day*, and v_2 is the view of *sales by month*, then v_1 is *finer* than v_2, since we can get the monthly sales by summing up the daily sales for the month. □

Our view selection algorithm is used for determining if a new view v is to be materialized for a query q, or if an existing materialized view m' should be used. We proceed with finding the best candidate m' from the materialised view set mv based on the calculation of b_{max}. If not found, v will be materialised if it meets the constraint S. Otherwise, we move on with calculating the benefit b_{sum} if materialising v will be more beneficial , $b_{sum} > s(v)$, for there may be OLAP views which can be computed using v. If $b_{sum} > s(v)$, we materialise v and indicate the best candidate from mv by m', that means, we will still compute v from m' in view maintenance along with the data warehouse.

Let us define the functions, mv, the materialised views; ov, the OLAP views; v, the view of query q with $vinov$; f, the frequency of OLAP view o; s, the size of view; and the-view:$ov \rightarrow mv$. Assume the space constraint S, and current usage of the space S', we define our view selection rule as follows:

```
$select(mv, ov, v(q), f(v), m') =$
   $b_{max} := 0, b_{sum} := 0$
   forall $m \in mv$ do
      if $m \succ v(q)$ then do
         $b(m) := (s(v(q)) - s(m)) \times f(v) + s(v(q))$;
         if $b(m) > b_{max}$ then
            $b_{max} = b(m), m' := m$ enddo
```

```
enddo
if $b_{max} = 0 \wedge S' + s(v(q)) \leq S$ then $mv := mv \cup v(q)$
else forall $o \in ov$ do
   if $v(q) \succ o$ then do
      $b(o) := (s(\text{the-view}(o)) - s(v)) \times f(o)$
      $b_{sum} := b_{sum} + b(o)$ enddo
enddo

if $b_{sum} > s(v) \wedge S' + s(v(q)) \leq S$ then $mv := mv \cup v(q)$
```

7.2.3 Application Cases

Example 7.2. Let us look at a case for view selection. Assume that our current data warehouse has two OLAP views:

View $\mathcal{V}_1$: the total sale by shop and day, its average number of tuples: $s(\mathcal{V}_1) = 20000$, its frequency $f(\mathcal{V}_1) = 30$;

View $\mathcal{V}_2$: the total sale by state and month, its average number of tuples: $s(\mathcal{V}_2) = 3000$, its frequency $f(\mathcal{V}_2) = 10$;

Assuming $\mathcal{V}_1$ is materialized, $\mathcal{V}_2$ is rewritten from $\mathcal{V}_1$ and not materialised, and space is not a concern in this case.

Now the user requests for a new OLAP query, view $\mathcal{V}$, the total sale by region and day, its average number of tuples: $s(\mathcal{V}) = 18000$, its frequency $f(\mathcal{V}) = 5$.

It is obvious that $\mathcal{V}_1 \succ \mathcal{V}$ holds, that means, we can rewrite the new view from $\mathcal{V}_1$. Our estimation of the benefit of rewriting $\mathcal{V}$ from $\mathcal{V}_1$:

$$\begin{aligned} b(\mathcal{V}_1) &= (s(\mathcal{V}) - s(\mathcal{V}_1)) \times f(\mathcal{V}) + s(\mathcal{V}) \\ &= (18000 - 20000) \times 5 + 18000 \\ &= 8000 \end{aligned}$$

As we have $b_{max} = b(\mathcal{V}_1)$ positive, we should rewrite $\mathcal{V}$ from $\mathcal{V}_1$. However, we shall also consider if materialising $\mathcal{V}$ will be more beneficial for other OLAP views which are rewritten from $\mathcal{V}_1$.

It is obvious that we have $\mathcal{V} \succ \mathcal{V}_2$, so the benefit of writing $\mathcal{V}_2$ from $\mathcal{V}$ is estimated:

$$\begin{aligned} b(\mathcal{V}_2) =& (s(\mathcal{V}_1) - s(mathcalV)) \times f(mathcalV_2) \\ &= (20000 - 18000) \times 10 \\ &= 20000 \end{aligned}$$

Now we have $b_{sum} = b(o)$ and $b_{sum} > s(\mathcal{V})$, that means, we should materialise the new view $\mathcal{V}$ and rewrite $\mathcal{V}_2$ from $\mathcal{V}$. □

Example 7.3. Let us look at a case of dynamic data warehouse design. Assume the money figures in the ground model are in US$, and we have an OLAP query on total sales in US$ with no figures on profit. A new OLAP query is requested from the store manager for total sales in EURO and the corresponding profit.

It is obvious that we are not able to rewrite the new view from the existing total sales with profit missing, so we extend the materialsied view set with the view for the new query. Let us define a function *cnv* for converting from EURO to US$. We execute the refinement steps as follows:

1. Add a new rule to the OLAP ASM:

```
$create\_V\_sales\_euro$ = forall $s, r, st, m, q, y$ with
    $\exists n, t, ph.(s, n, t, r, st, ph) \in \text{Shop} \wedge$
    $\exists d, d', w.(d, d', w, m, q, y) \in \text{Time}$
    do let $S = src[0, \pi_{s'}, +]$
            $(\{(i, s, p, d, s') \mid \exists q', p'.$
            $(i, s, p, d, q', s', p') \in \text{Purchase} \wedge$
            $d.month = m \wedge d.year = y\})$
        $P = src[0, \pi_{p'}, +]$
            $(\{(i, s, p, d, p') \mid \exists q', s'.$
            $(i, s, p, d, q', s', p') \in \text{Purchase} \wedge$
            $d.month = m \wedge d.year = y\})$
    in $\text{V\_sales\_euro}(s, r, st, m, q, y, cnv^{-1}(S), cnv^{-1}(P)) := 1$
  enddo
```

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 13, to add the OLAP view *V_Msales_euro* to the OLAP-ASM:

$$\text{OLAP-ASM}^* \rhd \text{V\_sales\_euro}$$
$$= shop \times region \times st \times month \times qtr \times year \times profit \times msale\_euro \to \{\mathbb{1}\}$$

3. Invoke view selection. As we have indicated we need to materialise the new view, and assume space is not a concern in this case, we proceed with adding the new view to the materialised view set in the next step.

4. Apply the schema transformation rule 13 to add the new OLAP view to the DW-ASM as a materialised view:

$$\text{DW-ASM}^* \rhd \text{MV\_V\_sales\_euro}$$
$$= shop \times region \times st \times month \times qtr \times year \times profit \times msale\_euro \to \{\mathbb{1}\}$$

5. Integrate controlled functions on the DW ASM: apply type integration *rule 18* to the materialised views as follows:

$$\frac{\begin{array}{ll}\text{DW-ASM} \rhd & \mathit{MV\_V\_sales\_euro} = \\ & \mathit{shop} \times \mathit{region} \times \mathit{st} \times \mathit{month} \times \\ & \mathit{qtr} \times \mathit{year} \times \mathit{msale\_euro} \times \mathit{profit} \rightarrow \{\mathbb{1}\} \\ & \mathit{MV\_V\_sales} = \\ & \mathit{shop} \times \mathit{region} \times \mathit{st} \times \mathit{month} \times \\ & \mathit{qtr} \times \mathit{year} \times \mathit{msale} \rightarrow \{\mathbb{1}\}\end{array}}{\begin{array}{ll}\text{DW-ASM}^* \rhd & \mathit{MV\_V\_sales\_prf} = \\ & \mathit{shop} \times \mathit{region} \times \mathit{st} \times \mathit{month} \times \\ & \mathit{qtr} \times \mathit{year} \times \mathit{profit} \times \mathit{msale} \rightarrow \{\mathbb{1}\}\end{array}} \varphi$$

Then the side condition φ is satisfied by:

- both of the types have the same key: $\mathit{sh} \times \mathit{month} \times \mathit{year}$;
- and they map to the same tuples populated from the same data warehouse instance with no additional selections;
- and the bijective mapping is defined as $h := (\mathit{id}, \mathit{id}, \mathit{id}, \mathit{id}, \mathit{cnv})$, where id is an identity function, for mapping the keys, and other three identical attributes, and cnv is the current currency conversion function from EURO to US\$ for mapping the total sales.

6. As a consequence of the integration, we need to replace the rule *refresh_MV_V_sales* at the DW-ASM by the following:

```
$\mathit{refresh\_MV\_V\_sales\_prf}$ =
        $\mathit{create\_MV\_V\_sales\_prf}$;
        forall $s, r, st, m, q, y, S, P$ with
           $(s, r, st, m, q, y, S, P) \in$ V_sales do
              MV_V_sales_prf$(s, r, st, m, q, y, S, P) := 1$ enddo
```

7. Similarly we replace the view creation rules *create_V_sales* and *create_V_sales_euro* at the OLAP-ASM by the followings:

```
$\mathit{create\_V\_sales\_prf}$ =
    forall $s, r, st, m, q, y$ with
       $\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
       $\exists d, d', w.(d, d', w, m, q, y) \in$ Time
       do let $S = src[0, \pi_{s'}, +]$
              $(\{(i, s, p, d, s') \mid \exists q', p'.$
              $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
              $d.month = m \wedge d.year = y\})$
          $P = src[0, \pi_{p'}, +]$
              $(\{(i, s, p, d, p') \mid \exists q', s'.$
              $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
```

```
            $d.month = m \wedge d.year = y\})$
        in $\text{V\_sales}(s, r, st, m, q, y, S, cnv^{-1}(P)) := 1$
    enddo
```

8. Then we need to change the datamart opening rules accordingly in DW-ASM:

```
    $open\_datamart(dm)$ = case the-matching-view($dm$) of
        V_sales : forall $s, r, st, m, q, y, S, P$ with
                $(s, r, st, m, q, y, S, P) \in$ MV_V_sales_prf do
                    DM-V_sales($dm, s, r, st, m, q, y, S$) := 1 enddo
        V_sales_euro : forall $s, r, st, m, q, y, S, P$ with
                $(s, r, st, m, q, y, S, P) \in$ MV_V_sales_prf do
                    DM-V_sales_euro($dm, s, r, st, m, q, y, cnv^{-1}(S), P$) := 1
                enddo
        endcase
```

□

7.3 Distribution Design

It is commonly accepted that operational databases and OLTP tasks are supported by distributed databases [81]. However, little work has been done on the distribution of OLAP tasks. For instance, it is quite possible that regional analysis of sales, profits, customer preferences, etc. will be executed regionally. Even a central data warehouse may receive its data only from local data warehouses. This reflects the distributed nature of modern organisations, the huge data volume processed for OLAP applications, and the non-acceptability of failure.

Obviously, as data warehouses support only read access in order to provide the data for the various data marts, a simple warehouse distribution could be obtained by full replication of the complete data warehouse. However, this is not efficient, as then a local data warehouse might contain a lot of data that is never used at that particular location. On the other hand, isolating exactly the data that is needed in local data warehouses may lead to an overly fragmented system.

Therefore, we propose to analyse and optimise query and maintenance costs. For this purpose we provide a simple query cost model, on the basis of which it is possible to reconsider a given fragmentation. The simple heuristic is that fragments may be re-combined, if this is advantageous from a performance point of view. A distribution grocery store data warehouse is given as an example in demonstration of design using the refinement-based design approach.

7.3.1 Architecture of Distributed Data Warehouses

As refreshing of the data warehouse content can be assumed to be executed in an off-line mode and the OLAP tier requires only read-access to the warehouse, we favour an architecture with as much replication as necessary, such that all view creation operations for data marts can be performed locally. This assumption is illustrated by the distributed data

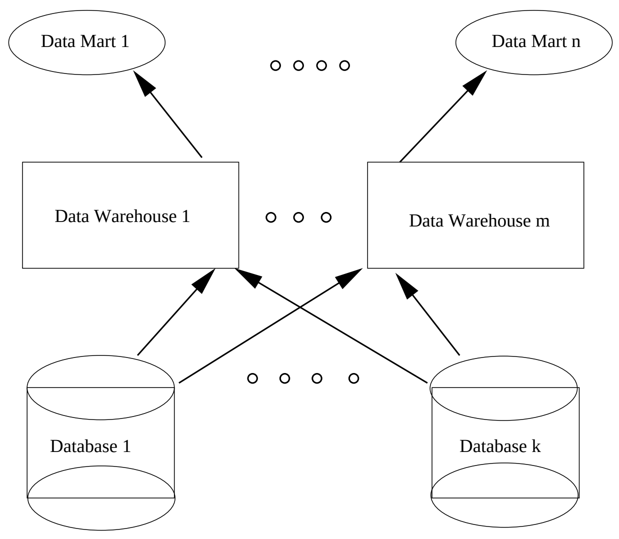

Figure 7.4: Distributed Data Warehouse Architecture

warehouse architecture in Figure 7.4. It is even possible that the whole data warehouse is replicated at all nodes.

For instance, in the sales example we used so far regional sales statistics may only be provided to regional branch offices, whereas a general sales statistics for a headquarter may disregard individual shops. This implies that different data marts would be needed for the locations participating in the distributed warehouse and OLAP system.

7.3.2 Fragmentation

The standard approach to distribution design for databases is to fragment the logical units, i.e. the relations in case of a relational database [81]. In fact, the way we modelled the data warehouse in Chapter 5 has a relational database schema at its core. We can therefore describe horizontal and vertical fragmentation as refinements of ASMs.

We may start with a fully replicated data warehouse, i.e. for each location ℓ we have a copy of DW-ASM as well as a tailored copy of OLAP-ASM, which only supports those data marts that are used at ℓ. Fragmenting the signature of DW-ASM impacts on the extraction rules defined on DW-ASM and the view creation rules on the copies of OLAP-ASM, whereas the local versions of DB-ASM remain unchanged. Therefore, without loss of generality we may concentrate on a single machine DW-ASM as the object of refinement.

As shown in Chapter 5, the signature of DW-ASM consists of a set of controlled functions S_i $(i = 1, \dots, k)$, each with an arity ar_i. Horizontal fragmentation of S_i with a selection formula φ leads to two new functions S_{i1} and S_{i2} with the same arity ar_i together with the constraints

$$S_{i1}(x_1, \dots, x_{ar_i}) \neq \bot \Leftrightarrow S_i(x_1, \dots, x_{ar_i}) \neq \bot \wedge \varphi(x_1, \dots, x_{ar_i})$$

and

$$S_{i2}(x_1, \dots, x_{ar_i}) \neq \bot \Leftrightarrow S_i(x_1, \dots, x_{ar_i}) \neq \bot \wedge \neg\varphi(x_1, \dots, x_{ar_i}).$$

In other words, the relation represented by S_i is the disjoint union of the relations represented by S_{i1} and S_{i2}. In DW-ASM we obtain the following additional rule:

```
extract_$S_{i1}(x)$ =
    extract_$S_i(y)$ ;
    forall $x_1, \dots, x_{ar_i}$ with $y(x_1, \dots, x_{ar_i}) = 1 \wedge \varphi(x_1, \dots, x_{ar_i}) = 1$
    do $x(x_1, \dots, x_{ar_i}) := 1$
    enddo
```

Similarly we obtain a rule extract_$S_{i2}(x)$. If fragment S_{ij} is used in DW-ASM, then this is reflected in an obvious change for refreshing.

Vertical fragmentation of S_i leads to n new functions $S_{i1}, \dots, S_{in}$ of arities $ar_{ij} < ar_i$ and subject to the constraints

$$S_{i1}(x_1, \dots, x_{ar_{ij}}) \neq \perp \Leftrightarrow \exists y_1, \dots, y_{ar_i}.S_i(y_1, \dots, y_{ar_i}) \wedge \bigwedge_{1 \le p \le ar_{ij}} x_p = y_{\sigma_j(p)}$$

for some injective $\sigma_j : \{1, \dots, ar_{ij}\} \to \{1, \dots, ar_i\}$ $(j = 1, \dots, n)$. In addition, we must have $\bigcup_{j=1}^{n} \{\sigma_j(1), \dots, \sigma_j(ar_{ij})\} = \{1, \dots, ar_i\}$, and the places $\sigma_j(1), \dots, \sigma_j(ar_{ij})$ must define a key for S_i. Thus, in DW-ASM we obtain the following additional rules:

```
extract_$S_{ij}(x)$ =
    extract_$S_i(y)$ ;
    forall $x_1, \dots, x_{ar_i}$ with $y(x_1, \dots, x_{ar_i}) = 1$
    do $x(x_{\sigma_j(1)}, \dots, x_{\sigma_j(ar_{ij})}) := 1$
    enddo
```

7.3.3 Query and Maintenance Cost

Fragmentation implies changes to the view creation rules in OLAP-ASM. In fact, these rules define the queries we are interested in. In developing a query cost model as in [73] we may assume that the queries are rewritten in a way that we first use a selection predicate φ to select tuples in a data warehouse relation S_i, then project them to some of the attributes. That is, each query involves subqueries of this form, which can be written as rules in OLAP-ASM in the following way

```
subquery($x$) =
    forall $x_1, \dots, x_{ar_i}$
    with $S_i(x_1, \dots, x_{ar_i}) = 1 \wedge \varphi(x_1, \dots, x_{ar_i}) = 1$
    do $x(x_{\sigma(1)}, \dots, x_{\sigma(q)}) := 1$
    enddo
```

for some S_i of arity ar_i in the signature of DW-ASM and an injective $\sigma : \{1, \dots, q\} \to \{1, \dots, ar_i\}$. The result of such a subquery corresponds to a fragment, first using horizontal fragmentation with φ, then vertical fragmentation with σ. It is a common heuristic to consider that some of these fragments may need to be re-combined to reduce the cost of maintenance. The costs of a query q can be composed as $c_1(q) + d \cdot \sum_{f \in \mathcal{F}(q)} s(f)$, where the sum ranges over the set $\mathcal{F}(q)$ of fragments f used in q, d is a constant, $s(f)$ is the size of fragment f, and $c_1(q)$ is independent of the fragmentation [74].

Thus, only $\sum_{f\in\mathcal{F}(q)} s(f)$ is influencing in the query costs of q in the fragmentation case. The query cost for the whole set of the queries at a site can be estimated by

$$\begin{aligned} qcosts &= \sum_q f_q \cdot \sum_{f\in\mathcal{F}(q)} s(f) \\ &= \sum_f \left(\sum_q n_{f,q} \cdot f_q \right) \cdot s(f) \\ &= \sum_f c(f) \cdot s(f) \ , \end{aligned}$$

where the first sum ranges over all queries, f_q denotes the frequency of query q, and $n_{f,q}$ denotes, the number of times the fragment f is used in computing the query q.

On the other hand, the maintenance costs correspond directly to the fragments built, i.e. we have

$$mcosts = d' \cdot \sum_f s(f)$$

with the sum ranging over all fragments used in the local version of DW-ASM and a constant d'. In combining query and maintenance costs we use a weighting factor, so the total costs can be rewritten in the form

$$\sum_f (c(f) + w) \cdot s(f)$$

with a constant w.

7.3.4 Recombination of Fragments

Of course, producing all fragments that are suggested by the queries produce minimal query costs. However, as fragments may overlap, the more fragments we use the higher the maintenance costs will be. Therefore, we should also consider the re-combination of fragments into a single fragment for some of the fragments. Suppose S_{i1} and S_{i2} are fragments derived from the function S_i using selection formulae φ and ψ and projections defined by σ and τ, respectively. We may assume without loss of generality that we can write $\sigma(j) = i_j$ and $\tau(j) = i_{x+j}$. Then the combined fragment $S_{i1} \uplus S_{i2}$ can be obtained from S_i by using selection with $\varphi \vee \psi$ followed by projection using $\sigma + \tau$, which is defined by

$$(\sigma + \tau)(j) = \begin{cases} \sigma(j) & \text{for } j = 1, \dots, k \\ \tau(j - x) & \text{else} \end{cases}$$

assuming σ is defined on $\{1, \dots, k\}$ and τ on $\{1, \dots, \ell\}$.

Alternatively, we may re-combine S_{i1} and S_{i2} by an outer-join, which would produce a result different from $S_{i1} \uplus S_{i2}$, in which irrelevant values are replaced by $\perp$. This can be produced by adding rules to DW-ASM of the form as below. Note that $y_1, \dots, y_l$ refer to attributes in $S_{i1} \setminus S_{i2}$, $k_1, \dots, k_m$ refer to attributes in $S_{i1} \cap S_{i2}$, which we assume to form a key for S_{i1} and S_{i2}, and $z_1, \dots, z_n$ refer to attributes in $S_{i2} \setminus S_{i1}$.

```
extract_$S_{i1} \oplus S_{i2}(X)$ =
    extract_$S_{i1}(Y)$ ; extract_$S_{i2}(Z)$ ;
    forall $k_1, \dots, k_m$ with
    $\exists y_1, \dots, y_l, z_1, \dots, z_n.(Y(y_1, \dots, y_l, k_1, \dots, k_m) = 1 \wedge$
      $Z(k_1, \dots, k_m, z_1, \dots, z_n) = 1)$ do
        let $y_1, \dots, y_l, z_1, \dots, z_n =$
        $\mathbf{I} y_1, \dots, y_l, z_1, \dots, z_n.(Y(y_1, \dots, y_l, k_1, \dots, k_m) = 1 \wedge$
          $Z(k_1, \dots, k_m, z_1, \dots, z_n) = 1)$ in
            $X(y_1, \dots, y_l, k_1, \dots, k_m, z_1, \dots, z_n) := 1$ enddo
forall $k_1, \dots, k_m$ with
  $\exists y_1, \dots, y_l.Y(y_1, \dots, y_l, k_1, \dots, k_m) = 1 \wedge$
    $\neg\exists z_1, \dots, z_n.Z(k_1, \dots, k_m, z_1, \dots, z_n) = 1$ do
      let $y_1, \dots, y_l = \mathbf{I} y_1, \dots, y_l.Y(y_1, \dots, y_l, k_1, \dots, k_m) = 1$ in
            $X(y_1, \dots, y_l, k_1, \dots, k_m, \perp, \dots, \perp) := 1$ enddo
forall $k_1, \dots, k_m$ with
  $\neg\exists y_1, \dots, y_l.Y(y_1, \dots, y_l, k_1, \dots, k_m) = 1 \wedge$
    $\exists z_1, \dots, z_n.Z(k_1, \dots, k_m, z_1, \dots, z_n) = 1$ do
      let $z_1, \dots, z_n = \mathbf{I} z_1, \dots, z_n.Z(k_1, \dots, k_m, z_1, \dots, z_n) = 1$ in
            $X(\perp, \dots, \perp, k_1, \dots, k_m, z_1, \dots, z_n) := 1$ enddo
```

With respect to the total costs we have $c(f_1 \oplus f_2) = c(f_1) + c(f_2)$ in the worst case, and max($c(f_1)$,$c(f_2)$) in the best case. Thus, we have to compare

$$costs = (c(f_1) + w) \cdot s(f_1) + (c(f_2) + w) \cdot s(f_2)$$

with

$$costs_{\text{new}} = (c(f_1 \oplus f_2) + w) \cdot s(f_1 \oplus f_2)$$

If we have $costs \geq costs_{\text{new}}$, it is advisable to re-combine the fragments f_1 and f_2, otherwise prefer to use f_1 and f_2. Obviously, a fragment can be combined with more than one other fragment, but it should appear in only one such combination. Thus, a non-deterministic process which selects pairs of fragments, compares costs and eventually combines them does not guarantee a global cost optimum.

7.3.5 Distribution design for a Grocery Store Data Warehouse

Distribution design is classified as implementation refinement in our ASM-based method. For ease of reference, we repeat the refinement steps as follows:

1. *Replicate the data warehouse and the OLAP ASMs*: for each node in the network assume the same copy of the data warehouse ASM and the OLAP ASM.

2. *Remove controlled functions and rules in local OLAP ASMs*: if the needed OLAP functionality is different at different network nodes, then these rules will simply reduce the corresponding OLAP ASM.

3. *Fragment controlled functions in local data warehouse ASMs*: this rule will reorganise and reduce a local data warehouse ASM, if the corresponding OLAP ASM does not need all of the replicated data warehouse. The refresh rules are then adapted accordingly.

4. *Recombine fragments in local data warehouse ASMs*: this rule will reorganise a local data warehouse ASM according to query cost considerations. The refresh rules are then adapted accordingly.

Example 7.4. Assume in the case of grocery store we have three locations, A, B, and C, each of them has a different analysis work such as: one looks after the daily sales of electronic products, with category ranging from 1000 inclusive to 2000; one monitors the daily quantity sold for products; and the other tracks the daily profit made by products sold.

Following the refinement steps in our method, we proceed as follows:

1. First we replicate the data warehouse and OLAP model at each node, which is to simply make three copies of the data warehouse models, and name them by sites.
2. Remove the unnecessary OLAP views and the rules, such that we have:

```
TASM OLAP-ASM$_a$
...
SIGNATURE
  V_sales_ele:$sid \times region \times state \times month \times quarter \times year \times$
        $sales \times profit \rightarrow \{\mathbb{1}\}$,
...
BODY
...
  $create\_V\_sales\_ele$ = forall $s, r, st, d, m, y, p$ with
      $\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
      $\exists d', w, q.(d, d', w, m, q, y) \in$ Time$\wedge$
      $\exists c, des.(p, c, des) \in$ Product $\wedge\ 1000 \leq c < 2000$
      do let $S = src[0, \pi_{s'}, +]$
            $(\{(i, s, p, d, s') \mid \exists q', p'.$
            $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
            $d.month = m \wedge d.year = y\})$
         $P = src[0, \pi_{p'}, +]$
            $(\{(i, s, p, d, p') \mid \exists q', s'.$
            $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
            $d.month = m \wedge d.year = y\})$
      in V_sales_ele$(s, r, st, m, q, y, S, P) := 1$
    enddo
...

TASM OLAP-ASM$_b$
...
SIGNATURE
  V_sales_qty:$sid \times region \times state \times date \times month \times year \times product$
    $\times quantity \rightarrow \{\mathbb{1}\}$,
...
BODY
...
  $create\_V\_sales\_qty$ = forall $s, r, st, d, m, y, p$ with
```

```
        $\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
        $\exists d', w, q.(d, d', w, m, q, y) \in$ Time$\wedge$
        $\exists c, des.(p, c, des) \in$ Product
        do let $Q = src[0, \pi_{q'}, +]$
                $(\{(i, s, p, d, q') \mid \exists s', p'.$
                $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
                $d.month = m \wedge d.year = y\})$
        in V_sales_qty$(s, r, st, d, m, y, p, Q) := 1$
    enddo

TASM OLAP-ASM$_c$
...
SIGNATURE
  V_sales_prf:$sid \times region \times state \times date \times month \times year \times product$
    $\times sales \times profit \rightarrow \{1\!\!1\},$
...
BODY
...
  $create\_V\_sales\_prf$ = forall $s, r, st, d, m, y, p$ with
        $\exists n, t, ph.(s, n, t, r, st, ph) \in$ Shop$\wedge$
        $\exists d', w, q.(d, d', w, m, q, y) \in$ Time$\wedge$
        $\exists c, des.(p, c, des) \in$ Product
        do let $S = src[0, \pi_{s'}, +]$
                $(\{(i, s, p, d, s') \mid \exists q', p'.$
                $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
                $d.month = m \wedge d.year = y\})$
            $P = src[0, \pi_{p'}, +]$
                $(\{(i, s, p, d, p') \mid \exists q', s'.$
                $(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
                $d.month = m \wedge d.year = y\})$
        in V_sales_prf$(s, r, st, d, m, y, p, S, P) := 1$
    enddo
```

3. Fragmentation in local data warehouse ASMs: it is obvious that tracking of electronic product sales requires a subset of the tuples in **Purchase**, for which horizontal fragmentation can be used, and tracking the quantity sold by product and the sales and profit by product sold require subsets of the columns in **Purchase**, which can be realised by vertical fragmentation.

   Let us start with the horizontal fragmentation. In our example we have predicates φ: $1000 \leq c' < 2000$ and ψ: $c' < 1000 \vee c' \geq 2000$. The horizontal fragmentation by φ and ψ gives us two fragments f_1 and f_2. Our vertical fragmentation will be carried out on f_1 and f_2 by two projections σ_1, which eliminates sales and profit the quantity, and σ_2, which eliminates the quantity. These give us fragments f_{11}, f_{12}, f_{21} and f_{22}.

   Based on the requirement, we have the fragments allocated as follows: f_{11} and f_{12} in

A, f_{11} and f_{21} in B, and f_{12} and f_{22} in C. Then we adapt the refresh rules accordingly:

```
TASM DW-ASM$_a$
...
SIGNATURE
  $\mathrm{f}_{11} : cid \times sid \times pid \times time \times qty \to \{\mathbb{1}\}$,
  $\mathrm{f}_{12} : cid \times sid \times pid \times time \times sales \times profit \to \{\mathbb{1}\}$,
  $\mathrm{temp}_{f_1} = cid \times sid \times pid \times time \times qty \times sales \times profit \to \{\mathbb{1}\}$,
  $\mathrm{temp}_{\mathrm{Purchase}} = cid \times sid \times pid \times date \times qty \times sale \times profit \to \{\mathbb{1}\}$,
...
BODY
...
  $extract\_f_{11} = extract\_f_1$;
    forall $c, s, p, t, Q, S, P$ with $(c, s, p, t, Q, S, P) \in f_1$
    do $\mathrm{f}_{11}(c, s, p, t, Q) := 1$ enddo

  $extract\_f_{12} = extract\_f_1$;
    forall $c, s, p, t, Q, S, P$ with $(c, s, p, t, Q, S, P) \in f_1$
    do $\mathrm{f}_{12}(c, s, p, t, S, P) := 1$ enddo

  $extract\_f_1 =$
    $extract\_purchase$;
    forall $c, s, p, t, Q, S, P$ with $(c, s, p, t, Q, S, P) \in \mathrm{Purchase} \wedge$
        $\exists c', d.(p, c', d) \in \mathrm{Product} \wedge 1000 \leq c' < 2000$
    do $\mathrm{temp}_{f_1}(c, s, p, t, Q, S, P) := 1$ enddo

  $extract\_purchase =$
    forall $i, p, s, d, p', c$ with
    $\exists t.(i, p, s, t, p', c) \in \pi_{cid,pid,sid,time,price,cost}$
        $(\mathrm{Buys} \bowtie \mathrm{Customer\_DB} \bowtie \mathrm{Part} \bowtie \mathrm{Store} \bowtie \mathrm{Offer}) \wedge t.date = d$
    do let $Q = src[0, \pi_q, +](\{(t, q) \mid (i, s, p, t, q) \in Buys \wedge$
        $t.date = d\})$, $S = Q * p'$, $P = Q * (p' - c)$
        in $\mathrm{temp}_{\mathrm{Purchase}}(i, p, s, d, Q, S, P) := 1$ enddo

TASM DW-ASM$_b$
...
SIGNATURE
  $\mathrm{f}_{11} : cid \times sid \times pid \times time \times qty \to \{\mathbb{1}\}$,
  $\mathrm{f}_{21} : cid \times sid \times pid \times time \times qty \to \{\mathbb{1}\}$,
  $\mathrm{temp}_{f_1} = cid \times sid \times pid \times time \times qty \times sales \times profit \to \{\mathbb{1}\}$,
  $\mathrm{temp}_{f_2} = cid \times sid \times pid \times time \times qty \times sales \times profit \to \{\mathbb{1}\}$,
  $\mathrm{temp}_{\mathrm{Purchase}} = cid \times sid \times pid \times date \times qty \times sale \times profit \to \{\mathbb{1}\}$,
...
BODY
...
  $extract\_f_{11} = \ldots$
  $extract\_f_{21} = extract\_f_2$;
    forall $c, s, p, t, Q, S, P$ with $(c, s, p, t, Q, S, P) \in f_2$
    do $\mathrm{f}_{21}(c,s,p,t,Q) := 1$ enddo
  $extract\_f_1 =\ldots$
  $extract\_f_2 =$
    $extract\_purchase$;
    forall $c,s,p,t,Q,S,P$ with $(c,s,p,t,Q,S,P) \in \mathrm{Purchase} \wedge$
        $\exists c', d.(p,c',d) \in \mathrm{Product} \wedge 1000 > c' \vee c' \geq 2000$
    do $\mathrm{temp}_{f_2}(c,s,p,t,Q,S,P) := 1$ enddo
  $extract\_purchase =\ldots$

TASM DW-ASM$_c$
...
SIGNATURE
  $\mathrm{f}_{12} : cid \times sid \times pid \times time \times sales \times profit \rightarrow \{1\!\!1\}$,
  $\mathrm{f}_{22} : cid \times sid \times pid \times time \times sales \times profit \rightarrow \{1\!\!1\}$,
  $\mathrm{temp}_{f_1} = cid \times sid \times pid \times time \times qty \times sales \times profit \rightarrow \{1\!\!1\}$,
  $\mathrm{temp}_{f_2} = cid \times sid \times pid \times time \times qty \times sales \times profit \rightarrow \{1\!\!1\}$,
  $\mathrm{temp}_{\mathrm{Purchase}} = cid \times sid \times pid \times date \times qty \times sale \times profit \rightarrow \{1\!\!1\}$,
; ...
BODY
...
  $extract\_f_{12} =\ldots$
  $extract\_f_{22} = extract\_f_2$;
    forall $c,s,p,t,Q,S,P$ with $(c,s,p,t,Q,S,P) \in f_2$
    do $\mathrm{f}_{22}(c,s,p,t,S,P) := 1$ enddo
  $extract\_f_1 =\ldots$
  $extract\_f_2 =\ldots$
  $extract\_purchase =\ldots$
```

4. Recombine fragments in local data warehouse ASMs: as we have suggested, for a better balanced query cost and redundancy, we should check if recombination of fragments is beneficial. To proceed with recombination, let us assume the following:

   - The average size of a tuple over Purchase as: $(64 \times 3 + 32 \times 3) = 672$(bits);
   - The average number of tuples in Purchase as 400;
   - The average size of a tuple over f_{11} is the same as over the other fragments, f_{12}, f_{21}, and f_{22}, as 640 bits;
   - The average number of tuples in f_{11} is the same as in f_{12}, as 100;
   - The average number of tuples in f_{21} is also the same as in f_{22}, as 300;
   - The size of f_{11} and f_{12} as $64,000$, and f_{21} and f_{22} as $192,000$;
   - The weighting factor w as 0.4;
   - The frequency $c(f_{11})$ and $c(f_{12})$ in location A as 30;
   - $c(f_{11})$ and $c(f_{21})$ in location B as 10;
   - $c(f_{12})$ and $c(f_{22})$ in location C as 20.

   So the total costs for not combining and recombining are estimated by:

In location A:

$$\begin{aligned} costs &= (c(f_{11}) + w) \cdot s(f_{11}) + (c(f_{12}) + w) \cdot s(f_{12}) \\ &= (30 + 0.4) \cdot 64000 + (30 + 0.4) \cdot 64000 \\ &= 3891200 \end{aligned}$$

$$\begin{aligned} costs_{\text{new}} &= (c(f_{11} \oplus f_{12}) + w) \cdot s(f_{11} \oplus f_{12}) \\ &= (c(f_{11}) + w) \cdot s(f_{11} \oplus f_{12}) \\ &= (30 + 0.4) \cdot 67200 \\ &= 2042880 \end{aligned}$$

In location B,

$$\begin{aligned} costs &= (c(f_{11}) + w) \cdot s(f_{11}) + (c(f_{21}) + w) \cdot s(f_{21}) \\ &= (10 + 0.4) \cdot 64000 + (10 + 0.4) \cdot 192000 \\ &= 2662400 \end{aligned}$$

$$\begin{aligned} costs_{\text{new}} &= (c(f_{11} \oplus f_{21}) + w) \cdot s(f_{11} \oplus f_{21}) \\ &= (c(f_{21}) + w) \cdot s(f_{11} \oplus f_{21}) \\ &= (10 + 0.4) \cdot 256000 \\ &= 2662400 \end{aligned}$$

In location C,

$$\begin{aligned} costs &= (c(f_{12}) + w) \cdot s(f_{12}) + (c(f_{22}) + w) \cdot s(f_{22}) \\ &= (20 + 0.4) \cdot 64000 + (20 + 0.4) \cdot 192000 \\ &= 5324800 \end{aligned}$$

$$\begin{aligned} costs_{\text{new}} &= (c(f_{12} \oplus f_{22}) + w) \cdot s(f_{12} \oplus f_{22}) \\ &= (c(f_{12}) + w) \cdot s(f_{12} \oplus f_{22}) \\ &= (20 + 0.4) \cdot 256000 \\ &= 5324800 \end{aligned}$$

Based on the estimation shown above, it is more cost effective to recombine fragments f_{11} and f_{12} at node A. The effect is to have f_1 at A. The refresh rules should again be adapted accordingly. As it is simple and straightforward, we omit the details.

□

7.4 Application of Business Statistics

The main undertaking of data warehousing is to support the business analysis happening in the OLAP tier. Most of such analysis is data intensive, and involves applying business statistics for supporting managers in their decision making. Often such requirements are dynamic and arise with the change of economic and business conditions. In the following, we take single linear regression, correlation and time series analysis as the examples in the dynamic data warehouse design to demonstrate how these changes are incorporated in the data warehouse process model using our tailored refinement method.

7.4.1 Single Linear Regression and Correlations

Regression analysis is used primarily for the purpose of prediction. The goal is to develop a statistical model that can be used to predict the values of a dependent variable based on the values of at least one independent variable. In contrast to regression, correlation analysis is used to measure the strength of the association between numerical variables [70].

Example 7.5. Let us take one of the examples in [70] for the case of grocery store data warehousing. We aim at showing how the ground model is refined to support the regression analysis. Our scenario is that the director of the grocery stores is being asked to develop an approach for forecasting annual sales for all new stores. Suppose he decided to examine the relationship between the size (square footage) of a store and its annual sales, that is, to build a sample linear regression model equation as follows:

$$\hat{Y}_i = b_0 + b_1 X_i$$

where

$$\begin{aligned} \hat{Y}_i &= \text{predicted value of } Y \text{ for observation } i \\ X_i &= \text{value of } X \text{ for observation } i \end{aligned}$$

In our example, the X is the size of a store, and the Y is the annual sales.

In order to predict the value Y, we need to compute the two coefficients, b_0 (the sample Y intercept) and b_1 (the sample slope). The simplest way is to use the least-squares method, which requires a sample with details of store size and annual sales from the data warehouse. Thus we need to make the required data, store size and annual sales available for the sample selection.

Following the refinement process described in Chapter 5, we refine the ground model for the requirement on data in the following:

1. Add a new rule in OLAP: we first define a rule *populate_V_size_sales* for populating the required data on store size and the annual sales.

```
$populate\_size\_sales$ = forall $s, size, y$ with
    $(s, size) \in \text{Store} \wedge$
    $\exists d, d', w, q, m.(d, d', w, m, q, y) \in \text{Time} \wedge$
    do let$S = src[0, \pi_{s'}, +]$
        $(\{(i, s, p, d, s') \mid \exists q', p'.(i, s, p, d, q', s', p') \in \text{Purchase} \wedge$
            $d.year = y\})$
        in V_size_sales$(s, size, y, S) := 1$
    enddo
```

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 13, to add function **V_size_sales**: $(sid \times size \times year \times sales)$ for supporting the above OLAP rule.
3. Add a new controlled function to DW ASM: as the store size is not available in the current data warehouse, we define a function **Store**$(sid \times size)$ in DW ASM.
4. Integrate controlled functions on the DW ASM: apply the schema transformation rule 18, the two functions **Shop** and **Store** are integrated to **Shop**$(sid \times name \times size \times town \times region \times state \times phone)$.
5. Change the view creation rules on the OLAP ASM: this is a consequence from view integration.

```
$populate\_size\_sales$ = forall $s, size, y$ with
    $\exists n, t, r, st, ph.(s, n, t, r, st, ph, size) \in$ Shop$\wedge$
    $\exists d, d', w, q, m.(d, d', w, m, q, y) \in$ Time$\wedge$
    do let$S = src[0, \pi_{s'}, +]$
        $(\{(i, s, p, d, s') \mid \exists q', p'.(i, s, p, d, q', s', p') \in$ Purchase $\wedge$
        $d.year = y\})$
    in V_size_sales$(s, size, y, S) := 1$
enddo
```

6. Change the rules in the DW ASM: as one of the consequences of integration, the refresh rule *extract_shop* will be changed to extract the details of store size into the relation **Shop**.

```
$extract\_shop$ = forall $s, n, s', a$ with Store$(s, n, s', a) \neq \bot$
 do let $t = a.town, r = a.region, st = a.state, ph = a.phone$
     in Shop$(s, n, s', t, r, st, ph) := 1$ enddo
```

7. Change the rules in the OLAP ASM: in addition, all the other rules that are referring to **Shop** (which is changed in the integration), such as *create_V_sales* in the ground model OLAP ASM, will be changed to include a function f as follows:

$$newRule = oldRule[Shop/f(Shop)]$$

where f is defined as

$$f(Shop) = \{(s, n, t, r, st, ph) \mid \exists s'.(s, n, s', t, r, st, ph) \in Shop\}$$

Once the newly defined view **View_size_sales** is populated, it can be used as the population for sample selection. With a sample, say last year's sales as an example, we may proceed with the regression and correlation analysis, which can be realised through applying relevant formulas over the selected sample. We will not discuss it further due to it involving only the application of statistical calculations.

7.4.2 Time Series Analysis

Regression analysis provides a useful methodology for managerial decision making. Similarly, the business forecasting methods applying the concept of time series analysis are used in the process of managerial planning and control. Time series forecasting methods involve the projection of future values of a variable based entirely on the past and present observations of that variable. As an example, we may make prediction of next year annual sales for a store based on its annual sales from year 1990 up to now.

Numerous methods applying time series analysis are devised for the purpose of forecasting. In the following we discuss how our data warehouse ground model is refined to include the exponential smoothing technique as an additional OLAP function. Exponential smoothing is not just being used for smoothing (providing impression of long-term movements) but also for obtaining short term (one period into the future) forecasts [70].

The formula for exponential smoothing is defined as follows:

$$E_i = WY_i + (1 - W)E_{i-1}$$

where

E_i = value being computed in time period i
E_{i-1} = value being computed in time period $i-1$
Y_i = observed value of the time series in period i
W = subjectively assigned weight or smoothing coefficient(where$0 < W < 1$)
$E_1 = Y_1$

Example 7.6. In our grocery store example, let us consider the exponential smoothing (e-smoothing in short), for forecasting store's annual sales. First we refine the ground model to include the data population for annual sales by store and year. This will be a simple refinement as follows:

1. *(Add a new rule in OLAP)* we define a rule *populate-annual-sales* for populating data on the annual sales by store and year.

```
$populate\text{-}annual\text{-}sales$ = forall $s, y$ with
    $\exists n, t, r, st, ph.\mathrm{Shop}(s, n, t, r, st, ph) = 1 \wedge$
    $\exists d, d', w, m, q.\mathrm{Time}(d, d', w, m, q, y) = 1$
    do let $S = src[0, \pi_{s'}, +](\{(i, s, p, d, s') \mid \exists q', p'.$
        $\mathrm{Purchase}(i, s, p, d, q', s', p') = 1 \wedge d.year = y\})$
    in $\mathrm{V\_annual\_esales}(s, y, S) := 1$
  enddo
```

2. Add a new controlled function to the OLAP ASM: apply the schema transformation rule 13, to add function **V_annual_esales** (*sid* × *year* × *sales*) for supporting the new rule.

It would be possible to just model the OLAP function e-smooth as the open-datamart operation, for which we would need to change the open-datamart rule to incorporate the computation of e-smoothed value. Another simpler way is to keep the open-datamart unchanged and make the e-smooth function as a new operation. In the latter case, we have the refinements under system implementation as follows:

1. First, we change the operation type function

   $$\textbf{o-type} : \textbf{op} \rightarrow \{open, close, quit, e - smooth\}$$

   to include the new OLAP function *e-smooth*;

2. In order to process the new type **e-smooth**, we need to change the rule *main* by adding:

   ```
   elsif o-type(op)= e-smooth then E-SMOOTH
   ```

3. To support the new rule, we define two ASM functions **the-weight**: **op** → **weight** for retrieving the weight from the operation, and **Pre-evalue**: **sid** × **year** → **evalue** for getting the previous year's e-smoothed value.

4. Then we add a new rule for the OLAP function *e-smooth*:

```
E-SMOOTH = if the-view(op) = V_annual_sales then
 import dm
   datamart(dm) := 1, owner(dm) := issuer(op),
   the-matching-view(dm) := the-view(op)
 end-import;
 populate-annual-sales;
 for all $s, y, S$ with V_annual_sales$(s, y, S) = 1$ do
   if pre-esales-value(s,y) $\neq 0$ then let
     $E$ = the-weight$(op) * S + (1 -$ the-weight$(op))\, *$ pre-esales-value$(s, y)$
     in DM-V_annual_sales$(dm, s, y, S, E) := 1$
   else let $E = S$ in DM-V_annual_sales$(dm, s, y, S, E) := 1$ endif
 enddo
```

Chapter 8

Conclusion

The Data warehouse and OLAP system design process is complex, lengthy, and error-prone due to its wide coverage of the business processes of the enterprise, and its dynamic nature of user requirements for the ever changing business conditions.

Current design methods mainly follow one of the two major approaches, i.e. the "top-down" approach [53], which builds up the data warehouse first, and caters for the user requirements in form of data marts later, and the "bottom-up" approach [58], which focuses on the data marts only, leaving the data warehouse virtual. Projects using the former approach can fail due to delaying "ROI" or lacking of fund, and projects using the latter approach can fail due to data inconsistency.

In view of the issues of the design process and the problems in the current approaches, we proposed a method that aims to resolving them within one unified framework. First of all, we model the entire system by the three-tier architecture, consisting of the data sources, the data warehouse and the OLAP system, while most methods focus on the data warehouse design only. We model both the data and the operations using ASMs, while most methods separate them by schema and process design. In order to have fast "ROI", we follow the "bottom-up" approach, but build the data warehouse at the same time. We resolve the inconsistency issue through schema integration.

We dealt with the issues of the design process in a number of ways. First of all, we applied the ASM-based method. The advantages of the ASM-based method are that it allows us to model a system at a level close to the domain of the system with clear operational semantics, so modelling work becomes simple for both the users and the designers. The correctness checking of the first model, i.e. the ground model, can be done by simply inspection. Formal verification is possible when the system properties are formalised. The ground model is then subject to stepwise refinements, which not only move the system from abstract to concrete, but also record precisely the intended design decisions at each design step.

Secondly we provided a refinement-based design framework, which breaks down the design process by design concerns, such as capturing new requirements, optimisation, and implementation. This is done using specific refinement steps for some typical design concerns in data warehouse and OLAP design, such as incremental data warehouse updates, view materialisation for query performance, data distribution design, etc.

Finally we ease the designers' work in the schema integration process by providing a pragmatic schema integration process with a set of provably correct schema transformation rules. This shifts the focus from formal refinement proofs to simple rule application.

Our idea of typed ASM is simpler than those in the literature. We aim to provide the well developed terms in database technology for data warehouse modelling, in particular schema integration.

Our future work includes exploring the possibility of using an execution tool to support the model validation, by translating the TASM for a chosen tool. Furthermore we plan to incorporate changes from the data sources, and cater for user requirements in the form of updates and deletions.

Index

Bibliography

[1] Abrial, J.-R. *The B-Book: Assigning Programs to Meanings.* Cambridge University Press, 1996.

[2] Adamson, C., and Venerable, M. *Data Warehouse Design Solutions.* Wiley Computer Publishing, 1998.

[3] Agerholm, S., and Larsen, P. G. A Lightweight Approach to Formal Methods. In *Proceedings of the International Workshop on Current Trends in Applied Formal Methods* (Boppard, Germany, 1998), Springer-Verlag.

[4] Agrawal, D., Abbadi, A. E., Singh, A., and Yurek, T. Efficient view maintenance at data warehouses. In *SIGMOD* (1997), pp. 417–427.

[5] Agrawal, R., Gupta, A., and Sarawagi, S. Modeling multidimensional database. In *Proc. Data Engineering Conference, Birmingham* (1997), pp. 232–243.

[6] Baralis, E., Paraboschi, S., and Teniente, E. Materialized views selection in a multidimensional database. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece* (1997), M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, Eds., Morgan Kaufmann, pp. 156–165.

[7] Barnett, M., Campbell, C., Schulte, W., and Veanes, M. Specification, simulation and testing of com components using abstract state machines. In *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001), Universidad de Las Palmas de Gran Canaria, Canary Islands, Spain* (2001), R. Moreno-Daz and A. Quesada-Arencibia, Eds., pp. 266–270.

[8] Biskup, J., and Convent, B. A formal view integration method. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data.* Association for Computing Machinery, 1986, pp. 398–407.

[9] Blaschka, M., Sapia, C., and Hoefling, G. On schema evolution in multidimensional databases. In *Data Warehousing and Knowledge Discovery – DaWaK'99* (1999), M. Mohania and A. M. Tjoa, Eds., vol. 1676 of *LNCS*, Springer-Verlag, pp. 153–164.

[10] Blass, A., and Gurevich, J. Abstract state machines capture parallel algorithms. *ACM Transactions on Computational Logic 4*, 4 (2003), 578–651.

[11] Boehm, B. W. A spiral model of software development and enhancement. *Computer 21*, 5 (1988), 61–72.

[12] Börger, E. Why use evolving algebras for hardware and software engineering? In *SOFSEM '95: Proceedings of the 22nd Seminar on Current Trends in Theory and Practice of Informatics* (London, UK, 1995), Springer-Verlag, pp. 236–271.

[13] Börger, E. High level system design and analysis using abstract state machines. In *Current Trends in Applied Formal Methods (FM-Trends 98)* (1999), D.Hutter, W.Stephan, P.Traverso, and M.Ullman, Eds., Springer LNCS, pp. 1–43.

[14] Börger, E. The ASM ground model method as a foundation for requirements engineering. In *Verification: Theory and Practice* (2003), pp. 145–160.

[15] Börger, E. The ASM refinement method. *Formal Aspects of Computing 15* (2003), 237–257.

[16] Börger, E., and Glässer, U. Modelling and analysis of distributed and reactive systems using evolving algebras. Tech. Rep. BRICS- NS-95- 4, University of Aarhus, 1995.

[17] Börger, E., and Stärk, R. *Abstract State Machines.* Springer-Verlag, Berlin Heidelberg New York, 2003.

[18] Börger, E., Stärk, R., and Schmid, J. *Java and the Java Virtual Machine: Definition, Verification and Validation.* Springer-Verlag, Berlin Heidelberg New York, 2001.

[19] Bouzeghoub, M., and Kedad, Z. A logical model for data warehouse design and evolution. In *DaWaK 2000: Proceedings of the Second International Conference on Data Warehousing and Knowledge Discovery* (London, UK, 2000), Springer-Verlag, pp. 178–188.

[20] Bowen, J. The Z notation. `http://vl.zuser.org/`.

[21] Bowen, J. The B-method, 2000. `http://vl.fmnet.info/b/`.

[22] Bowen, J., and Reeves, S. Formal models for informal gui designs. *Electron. Notes Theor. Comput. Sci. 183* (2007), 57–72.

[23] Bowen, J. P., and Hinchey, M. G. Ten commandments of formal methods. *IEEE Computer 28*, 4 (1995), 56–63.

[24] Chandra, A., and Harel, D. Computable queries for relational data bases. *Journal of Computer and System Sciences 21* (1980).

[25] Cunningham, C., Song, I.-Y., and Chen, P. P. Data warehouse design to support customer relationship management analyses. In *DOLAP '04: Proceedings of the 7th ACM international workshop on Data warehousing and OLAP* (New York, NY, USA, 2004), ACM, pp. 14–22.

[26] Dahl, O. J., Dijkstra, E. W., and Hoare, C. A. R., Eds. *Structured programming.* Academic Press Ltd., London, UK, UK, 1972.

[27] de Roever, W.-P., and Engelhardt, K. *Data Refinement: Model-Oriented Proof Methods and their Comparison.* Cambridge University Press, 1998.

[28] Del Castillo, G., Gurevich, Y., and Stroetmann, K. Typed abstract state machines. unpublished, available from http://research.microsoft.com/ gurevich/Opera/137.pdf, 1998.

[29] Derrick, J., and Boiten, E. *Refinement in Z and object-Z: foundations and advanced applications*. Springer-Verlag, London, UK, 2001.

[30] Diller, A. *Z: An Introduction to Formal Methods*. Wiley, 1994.

[31] DMReview.com. DW Basics Channel, 2008. `http://www.dmreview.com/channels/dw_basics.html`.

[32] Fitzgerald, J. The VDM Portal. `http://www.vdmportal.org/twiki/bin/view`.

[33] Fitzgerald, J., and Larsen, P. G. *Modelling Systems: Practical Tools and Techniques for Software Developmen*. Cambridge University Press, 1998.

[34] Golfarelli, M., Maio, D., and Rizzi, S. Conceptual design of data warehouses from E/R schema. In *HICSS '98: Proceedings of the Thirty-First Annual Hawaii International Conference on System Sciences-Volume 7* (Washington, DC, USA, 1998), IEEE Computer Society, p. 334.

[35] Golfarelli, M., and Rizzi, S. Methodological framework for data warehouse design. In *International Workshop on Data Warehousing and OLAP* (1998), pp. 3–9.

[36] Griffin, T., and Libkin, L. Incremental maintenance of views with duplicates. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995* (1995), M. J. Carey and D. A. Schneider, Eds., ACM Press, pp. 328–339.

[37] Gupta, A., Mumick, I. S., and Ross, K. A. Adapting materialized views after redefinitions. In *ACM SIGMOD* (1995), pp. 211–222.

[38] Gupta, A., Mumick, I. S., and Subrahmanian, V. S. Maintaining views incrementally. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data, Washington, D.C., May 26-28, 1993* (1993), P. Buneman and S. Jajodia, Eds., ACM Press, pp. 157–166.

[39] Gupta, H. Selection of views to materialize in a data warehouse. In *ICDT* (1997), pp. 98–112.

[40] Gupta, H., Harinarayan, V., Rajaraman, A., and Ullman, J. D. Index selection for OLAP. In *ICDE* (1997), pp. 208–219.

[41] Gupta, H., and Mumick, I. S. Selection of views to materialize under a maintenance cost constraint. *Lecture Notes in Computer Science 1540* (1999), 453–470.

[42] Gurevich, Y. Evolving algebras: An attempt to discover semantics. In *Current Trends in Theoretical Computer Science*, G. Rozenberg and A. Salomaa, Eds. World Scientific, River Edge, NJ, 1993, pp. 266–292.

[43] Gurevich, Y. Evolving algebras 1993: Lipari Guide. In *Specification and Validation Methods*, E. Börger, Ed. Oxford University Press, 1994, pp. 9–37.

[44] Gurevich, Y. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic 1*, 1 (2000), 77–111.

[45] Gurevich, Y., Sopokar, N., and Wallace, C. Formalizing database recovery. *Journal of Universal Computer Science 3*, 4 (1997), 320–340.

[46] Hall, A. Correctness by construction. `http://www.anthonyhall.org/html/technology.html`.

[47] Hall, A. Seven myths of formal methods. *IEEE Softw. 7*, 5 (1990), 11–19.

[48] Harinarayan, V., Rajaraman, A., and Ullman, J. D. Implementing data cubes efficiently. In *Proceedings of ACM SIGMOD '96, Montreal, June 1996.*

[49] Huggins, J. Abstract state machines, 1996. `http://www.eecs.umich.edu/gasm/`.

[50] Hull, R. Relative information capacity of simple relational database schemata. *SIAM Journal of Computing 15*, 3 (1986), 856–886.

[51] Husemann, B., Lechtenborger, J., and Vossen, G. Conceptual data warehouse modeling. In *Design and Management of Data Warehouses* (2000), p. 6.

[52] Inmon, B. Data mart does not equal data warehouse. *DM Direct Newsletter* (1999).

[53] Inmon, W. *Building the Data Warehouse.* Wiley & Sons, New York, 1996.

[54] Jacobson, I., Booch, G., and Rumbaugh, J. *The unified software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[55] Jacobson, I., Christerson, M., Jonsson, P., and Övergaard, G. *Object-oriented Software Engineering: A Use-Case Driven Approach.* Addison-Wesley, 1992.

[56] Jones, C. Systematic software development using VDM. `http://www.vdmportal.org/twiki/bin/view/Main/Jonesbook`.

[57] Kedad, Z., and Métais, E. Dealing with semantic heterogeneity during data integration. In *Conceptual Modeling – ER'99* (1999), J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, Eds., vol. 1728 of *LNCS*, Springer-Verlag, pp. 325–339.

[58] Kimball, R. *The Data Warehouse Toolkit.* John Wiley & Sons, 1996.

[59] Kimball, R. A dimensional modeling manifesto. *DBMS 10*, 9 (1997), 58–70.

[60] Koh, J., and Chen, A. Integration of heterogeneous object schemas. In *Entity-Relationship Approach - ER'93*, R. Elmasri, V. Kouramajian, and B. Thalheim, Eds., vol. 823 of *LNCS*. Springer-Verlag, 1994, pp. 297–314.

[61] Kotidis, Y., and Roussopoulos, N. A case for dynamic view management. *ACM Trans. Database Syst. 26*, 4 (2001), 388–423.

[62] Labio, W. J., Zhuge, Y., Wiener, J. L., Gupta, H., García-Molina, H., and Widom, J. The WHIPS prototype for data warehouse creation and maintenance. In *SIGMOD* (1997), pp. 557–559.

[63] Larson, J., Navathe, S. B., and Elmasri, R. A theory of attribute equivalence in databases with application to schema integration. *IEEE Transactions on Software Engineering 15*, 4 (1989), 449–463.

[64] Lawrence, M., and Rau-Chaplin, A. Dynamic view selection for OLAP. In *DaWaK* (2006), pp. 33–44.

[65] Lee, K. Y., Son, J. H., and Kim, M.-H. Efficient incremental view maintenance in data warehouses. In *CIKM* (2001), pp. 349–357.

[66] Lehmann, T. *Ein pragmatisches Vorgehenskonzept zur Integration und Kooperation von Informationssystemen*. PhD thesis, TU Clausthal, 1999.

[67] Lehmann, T., and Schewe, K.-D. A pragmatic method for the integration of higher-order Entity-Relationship schemata. In *Conceptual Modeling - ER 2000*, A. H. F. Laender, S. W. Liddle, and V. C. Storey, Eds., vol. 1920 of *LNCS*. Springer-Verlag, 2000, pp. 37–51.

[68] Lehner, W., Albrecht, J., and Wedekind, H. Normal forms for multidimensional databases. In *SSDBM* (1998), pp. 63–72.

[69] Lenz, H.-J., and Shoshani, A. Summarizability in OLAP and statistical data bases. In *Statistical and Scientific Database Management* (1997), pp. 132–143.

[70] Levine, D., Krehbiel, T., and Berenson, M. *Business Statistics: A First Course*. Prentice-Hall, New Jersey, 2000.

[71] Lewerenz, J., Schewe, K.-D., and Thalheim, B. Modelling data warehouses and OLAP applications using dialogue objects. In *Conceptual Modeling – ER'99*, J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, Eds., vol. 1728 of *LNCS*. Springer-Verlag, 1999, pp. 354–368.

[72] Luján-Mora, S., and Trujillo, J. A comprehensive method for data warehouse design. In *DMDW* (2003).

[73] Ma, H., and Schewe, K.-D. A heuristic approach to horizontal fragmentation in object oriented databases. In *Proceedings of the 2004 Baltic Conference on Databases and Information Systems* (Riga, Latvia, 2004).

[74] Ma, H., and Schewe, K.-D. Query cost analysis for horizontally fragmented complex value databases. In *Proceedings of the Eighth Conference on Advances in Databases and Information Systems* (Budapest, Hungary, 2004).

[75] Ma, H., Schewe, K.-D., Thalheim, B., and Zhao, J. View integration and cooperation in databases, data warehouses and web information systems. *Journal on Data Semantics IV* (2005), 213–249.

[76] Mazón, J.-N., Trujillo, J., Serrano, M., and Piattini, M. Applying MDA to the development of data warehouses. In *DOLAP* (2005), pp. 57–66.

[77] Moody, D. L., and Kortink, M. A. R. From enterprise models to dimensional models: a methodology for data warehouse and data mart design. In *Design and Management of Data Warehouses* (2000), p. 5.

[78] Morgan, C. *Programming from specifications (2nd ed.).* Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1994.

[79] Morzy, T., and Wrembel, R. Modeling a multiversion data warehouse: A formal approach. In *ICEIS (1)* (2003), pp. 120–127.

[80] Mumick, I. S., Quass, D., and Mumick, B. S. Maintenance of data cubes and summary tables in a warehouse. In *SIGMOD* (1997), pp. 100–111.

[81] Özsu, T., and Valduriez, P. *Principles of Distributed Database Systems.* Prentice-Hall, 1999.

[82] Pedersen, T. B., and Jensen, C. S. Multidimensional data modeling for complex data. In *ICDE* (1999), pp. 336–345.

[83] Prat, N., Akoka, J., and Comyn-Wattiau, I. A UML-based data warehouse design method. *Decis. Support Syst. 42*, 3 (2006), 1449–1473.

[84] Prinz, A., and Thalheim, B. Operational semantics of transactions. In *Database Technologies 2003: Fourteenth Australasian Database Conference* (2003), K.-D. Schewe and X. Zhou, Eds., vol. 17 of *Conferences in Research and Practice of Information Technology*, pp. 169–179.

[85] pUML Group. The precise UML group. `http://www.cs.york.ac.uk/puml/`.

[86] Qian, X. Correct schema transformations. In *Advances in Database Technology - EDBT'96*, P. M. G. Apers, M. Bouzeghoub, and G. Gardarin, Eds., vol. 1057 of *LNCS*. Springer-Verlag, 1996, pp. 114–126.

[87] Quass, D. Maintenance expressions for views with aggregation. In *VIEWS* (1996), pp. 110–118.

[88] Reeves, S., and Streader, D. Stepwise refinement of processes. *Electronic Notes in Theoretical Computer Scienece 160* (2006), 275–289.

[89] Rizzi, S., Abelló, A., Lechtenbörger, J., and Trujillo, J. Research in data warehouse modeling and design: dead or alive? In *DOLAP* (2006), pp. 3–10.

[90] Rumbaugh, J., Jacobson, I., and Booch, G. *The Unified Modeling Language Reference Manual.* Addison-Wesley, 1999.

[91] Sapia, C., Blaschka, M., Hoefling, G., and Dinter, B. Extending the E/R model for the multidimensional paradigm. vol. 1552 of *LNCS*, Springer, pp. 105–116.

[92] Schellhorn, G. *Verification of Abstract State Machines.* PhD thesis, University of Ulm, 1999.

[93] Schellhorn, G. Verification of ASM refinements using generalized forward simulation. *j-jucs 7*, 11 (2001), 952–979.

[94] Schellhorn, G. ASM refinement and generalizations of forward simulation in data refinement: a comparison. *Theor. Comput. Sci. 336*, 2-3 (2005), 403–435.

[95] Schellhorn, G. ASM refinement preserving invariants. In *Proc. 14th International Workshop on Abstract State Machines – ASM 2007* (Grimstad, Norway, 2007), A. Prinz, Ed.

[96] Schellhorn, G. Completeness of ASM refinement. In *Proceedings of REFINE 2008* (2008), ENTCS. to appear.

[97] Schewe, K.-D. Specification and development of correct relational database programs. Tech. rep., Clausthal Technical University, Germany, 1997.

[98] Schewe, K.-D. UML: A modern dinosaur? – a critical analysis of the unified modelling language. In *Information Modelling and Knowledge Bases XII*, H. Jaakkola, H. Kangassalo, and E. Kawaguchi, Eds., Frontiers in Artificial Intelligence and Applications. IOS Press, Amsterdam, 2001, pp. 185–202.

[99] Schewe, K.-D., and Schewe, B. Integrating database and dialogue design. *Knowledge and Information Systems 2*, 1 (2000), 1–32.

[100] Schewe, K.-D., and Zhao, J. Typed abstract state machines for data-intensive applications. *Journal Knowledge and Information Systems 15*, 3 (2008), 381–391.

[101] Sciore, E., Siegel, M., and Rosenthal, A. Using semantic values to facilitate interoperability among heterogeneous information systems. *ACM TODS 19*, 2 (1994), 254–290.

[102] Sen, A., and Sinha, A. P. A comparison of data warehousing methodologies. *Commun. ACM 48*, 3 (2005), 79–84.

[103] Simitsis, A. Mapping conceptual to logical models for ETL processes. In *DOLAP '05: Proceedings of the 8th ACM international workshop on Data warehousing and OLAP* (New York, NY, USA, 2005), ACM Press, pp. 67–76.

[104] Spaccapietra, S., and Parent, C. View integration – a step forward in solving structural conflicts. *IEEE Transactions on Knowledge and Data Engineering 6*, 2 (1994), 258–274.

[105] Spivey, J. M. *Understanding Z: a specification language and its formal semantics.* Cambridge University Press, New York, NY, USA, 1988.

[106] Thalheim, B. *Entity-Relationship Modeling: Foundations of Database Technology.* Springer-Verlag, 2000.

[107] Theodoratos, D. Detecting redundancy in data warehouse evolution. In *Conceptual Modeling – ER'99* (1999), J. Akoka, M. Bouzeghoub, I. Comyn-Wattiau, and E. Métais, Eds., vol. 1728 of *LNCS*, Springer-Verlag, pp. 340–353.

[108] Theodoratos, D., Dalamagas, T., Simitsis, A., and Stavropoulos, M. A randomized approach for the incremental design of an evolving data warehouse. In *Proceedings of the 20th International Conference on Conceptual Modeling: Conceptual Modeling*, vol. 2224 of *LNCS*. Springer-Verlag, 2001, pp. 325–338.

[109] Theodoratos, D., and Sellis, T. Dynamic data warehouse design. In *Data Warehousing and Knowledge Discovery – DaWaK'99*, M. Mohania and A. M. Tjoa, Eds., vol. 1676 of *LNCS*. Springer-Verlag, 1999, pp. 1–10.

[110] Todman, C. *Designing a Data Warehouse: Supporting Customer Relationship Management.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2000.

[111] Tryfona, N., Busborg, F., and Christiansen, J. G. B. starER: A conceptual model for data warehouse design. In *International Workshop on Data Warehousing and OLAP* (1999), pp. 3–8.

[112] Tsois, A., Karayannidis, N., and Sellis, T. K. MAC: Conceptual data modeling for OLAP. In *Design and Management of Data Warehouses* (2001), p. 5.

[113] Turull Torres, J. M. On the expressibility and computability of untyped queries. *Annals of Pure and Applied Logic 108*, 1-3 (2001), 345–371.

[114] uni trier.de. Integrated formal methods.
`http://www.informatik.uni-trier.de/ ley/db/conf/ifm/index.html`.

[115] Van den Bussche, J. *Formal Aspects of Object Identity in Database Manipulation.* PhD thesis, University of Antwerp, 1993.

[116] Vassiliadis, P. Modeling multidimensional databases, cubes and cube operations. In *SSDBM '98: Proceedings of the 10th International Conference on Scientific and Statistical Database Management* (Washington, DC, USA, 1998), IEEE Computer Society, pp. 53–62.

[117] Widom, J. Research problems in data warehousing. In *Proceedings of the 4th International Conference on Information and Knowledge Management* (1995), ACM.

[118] Wikipedia. B-method. `http://en.wikipedia.org/wiki/B-Method`.

[119] Wikipedia. Formal methods. `http://en.wikipedia.org/wiki/Formal_methods`.

[120] Wikipedia. Software development process.
`http://en.wikipedia.org/wiki/Software_development_process`.

[121] Wikipedia. VDM - Industry Experience.
`http://en.wikipedia.org/wiki/Vienna_Development_Method`.

[122] Wikipedia. Z notation. `http://en.wikipedia.org/wiki/Z_notation`.

[123] Wirth, N. Program development by stepwise refinement. *Commun. ACM 14*, 4 (1971), 221–227.

[124] Zamulin, A. V. Typed Gurevich machines revisited. *Joint Bulletin of NCC and IIS on Computer Science 5* (1997), 1–26.

[125] Zhuge, Y., Garcia-Molina, H., Hammer, J., and Widom, J. View maintenance in a warehousing environment. In *SIGMOD Conference* (1995), pp. 316–327.