VIEW-CONSTRAINT DUALITY IN DATABASES AND SYSTEMS ENGINEERING

John Springer

Submitted to the faculty of the Graduate School

in partial fulfillment of the requirements

for the degree

Doctor of Philosophy

in the Department of Computer Science

Indiana University

August 2007

Accepted by the Graduate Faculty, Indiana University, in partial fulfillment of the requirements of the degree of Doctor of Philosophy.

Doctoral Committee

Edward L. Robertson, Ph.D.
(Principal Advisor)

Dennis P. Groth, Ph.D.

Catharine Wyss, Ph.D.

August 9, 2007

Steven D. Johnson, Ph.D.

To Kate and Andrew for your faith, hope, and love

Acknowledgments

How does one express the indescribable sense of gratitude that results from the immense contributions of so many? Space and time do not permit such an effort, and yet that is precisely the extent to which I am indebted to all who have supported me throughout this process. The list is indeed lengthy.

I begin with my research committee, without whom this research would not be possible. First and foremost, I wish to thank my principal advisor, Ed Robertson. Only with Ed's generous guidance and assistance does this research even exist; I truly stand on the shoulders of a giant. In addition, I want to thank Dennis Groth for his mentoring and direction throughout my entire graduate student experience. I also wish to thank Cathy Wyss for helping to instill in me a keen interest in query languages and serving as an inspiration. In addition, I thank Steve Johnson for motivating me to seek new knowledge.

I am very thankful to have made numerous friends in the Database Group, the

Computer Science Department, and the School of Informatics as well as throughout Indiana University and the city of Bloomington. It may not occur to you how instrumental you have been in helping me realize this dissertation, but I certainly recognize your importance in this effort. Thank you for your support and friendship.

In addition, I wish to acknowledge the impact that my undergraduate *alma mater*, Taylor University, has had on me. In particular, I want to thank Dr. Leon Adkison and Coach Paul Patterson for their lasting influences.

I have been the recipient of deep and abiding friendships, and I sincerely appreciate their steadfast encouragement and support. The Parkers, the Holtmanns, the Perkins, the Pritchards, Timmy T., BK, Steve B., Lobo...the list grows each day.

Mom and Dad, all that I am and do is a footnote to you; thank you for your love and sacrifices for me.

Tony, Mike, Debbie, Beth, and your families, thank you for never being rivals and always being good to me.

Pat and Bob, your acceptance of me into your family has truly been amazing; thank you for all that you do for me and my family.

Jon, TC, Jim, Krysha, Caroline, Samuel, Grace, and Elizabeth, thanks for demonstrating to me that goodness and generosity run in the family.

Andrew, may the only constraints on your view be goodness, grace, and truth.

Kate, this dissertation is in effect a love letter that you've written to me, because it testifies to the lengths that you will go to support me. I thank you with all of my heart.

Finally, to the One in whom I live and move and have my being, the *Sine Qua Non*, Yours is the glory and praise.

Abstract

In the database systems and systems engineering domains, the concepts of constraints and views are commonly and effectively used. Considered as highly distinct, they stand as well-established notions in both domains' bodies of knowledge. Constraints may be expressed as "business rules" independent of underlying models. In these situations, the correctness of the implementation of the constraint may be in jeopardy and potential development inefficiencies may result as well. Furthermore, the modeler or designer may employ a model or methodology that does not support natively the specification of a particular constraint. As a result, such constraints only manifest themselves in the information system, and consequently, one is only aware when the constraint has been violated.

The focus of this research is that in fact a duality does exist between views and constraints (hypothesis 1) and that this duality is a useful tool in the development of information systems (hypothesis 2). Employing both proof and empirical evidence, our investigation reveals that the accuracy with which the first hypothesis holds

depends upon the degree to which the constraints can be formalized. In the case of the relational data model, the constraints can be formalized. This is even true of the *semantic* constraints that are expressible in relational algebra and relational calculus. In the case of the other models that we explore, the constraints have a less formal expression, and views prove to be a method for interjecting more formality into the expression of the constraints.

Our results concerning the accuracy of the first hypothesis also hold for the second hypothesis; that is, the accuracy with which the second hypothesis holds depends upon the degree to which the constraints can be formalized. In the case of the relational data model, the constraints can be formalized, and thus we can generate general solutions for them. Nevertheless, uses of the duality outside the relational domain may be possible, and this is a focus of future work.

Contents

List of Tables

List of Figures

1

Introduction

In the database systems and systems engineering domains, the concepts of constraints and views are commonly and effectively used. Considered as highly distinct, they stand as well-established notions in both domains' bodies of knowledge. The focus of this dissertation is that in fact a duality does exist between views and constraints and that this duality is a useful tool in the development of information systems.

As currently deployed, constraints may be expressed as "business rules" independent of underlying models – the one notable exception being cardinality constraints in Entity-Relationship (ER) models. Moreover, constraints generally have a local flavor, in that they are defined and evaluated with respect to a local context.[1] In these situations, the constraints typically find their first and only precise representation in

[1] As an example of the global-local distinction from the physical database realm, contrast tuples – which are local – with databases.

the source code, and this *ad hoc* approach tends to jeopardize the correctness of the implementation of the constraint.

Only slightly less troubling are the potential development inefficiencies resulting from the use of the *ad hoc* tactics. These inefficiencies arise from the inability to see commonly shared patterns and procedures, and this myopic perspective manifests itself in solutions best described as one-off. The inefficiencies are not only in the implementation time and effort but also in the performance of the implementation. More specifically, we see the absence of the value gained by having and utilizing common procedures that undergo heavy algorithmic scrutiny.

In general, the modeler and designer may be unable to fully and precisely express constraints where they arise, so they are "baked" into the information system. That is, the modeler or designer may employ a model or methodology that does not support natively the specification of a particular constraint, where the constraint itself is most naturally expressed in the context provided by the model/methodology. For example, consider the case where a functional constraint (that is, business rule) applies most naturally to the entities in an ER model, but the conventional ER notation does not support the specification of such a constraint. As a result, such constraints only manifest themselves in the information system, and consequently, one is only aware when the constraint has been violated, in effect recording the failure rather than catching the failure. We denote instances where constraints do not have native

support as cases of constraint specification displacement.

A View is an organized subset of a whole, where the subset is chosen according to some criteria. Furthermore, the term "view" usually denotes an external representation meant to conceal the internal structure of the whole and display only the information pertinent to the intended viewer. Moreover, a view typically has a global nature (that is, one usually defines them in a global manner). By virtue of this, views often have a distinctly different flavor than that possessed by constraints.

However, although the modeling methodologies in the database systems and systems engineering domains share roughly the same notion of view, each naturally has its own method for expressing views; and differences in these methods create a type of impedance mismatch, as views expressed according to one methodology do not interact naturally with views expressed according to another. For example, in the case of database systems (specifically, relational database management systems), a well-accepted method for expressing views does exist, but this Coddian mechanism tends to have limited feasibility outside its provenance. A shared mechanism for expressing views would certainly enable the expression of richer models.

The risky *ad hoc* implementation of constraints and the absence of a shared view mechanism across the database systems and systems engineering domains present compelling exigencies, and this research offers a solution to fill these needs. The remainder of this chapter describes a motivating example for this research and then

provides an overview of the document, including an outline of the hypotheses central to the research.

1.1 A Motivating Example

First consider a scenario involving truck assembly for the fictional automobile manufacturer Hanyoda Motors. Hanyoda's assembly process for its Farce trucks includes activities in the following physical areas on the assembly line: body shop, paint shop, trim, chassis, and final line. In the body shop, the truck's assembly begins with the assembly of its framework as well as its outer shell. Next, the assembly process proceeds through the paint shop and trim area; the truck's cab, box, and chassis remain separate. In parallel come the activities in the chassis area where the truck's skeleton receives many critical components, including its transmission and previously assembled engine. The entire process culminates in the joining of the cab and box to the chassis in the final line.[2]

To help meet the informational needs related to the truck assembly process, Hanyoda Motors' data modelers created entity-relationship (ER) models to depict the process' data requirements. Figure 1.1 contains the ER model that strictly concerns the parts needed by Hanyoda Motors in its truck assembly process.

[2]Please review [Cor] for a description of an actual truck assembly process; the process described in this document loosely mirrors it.

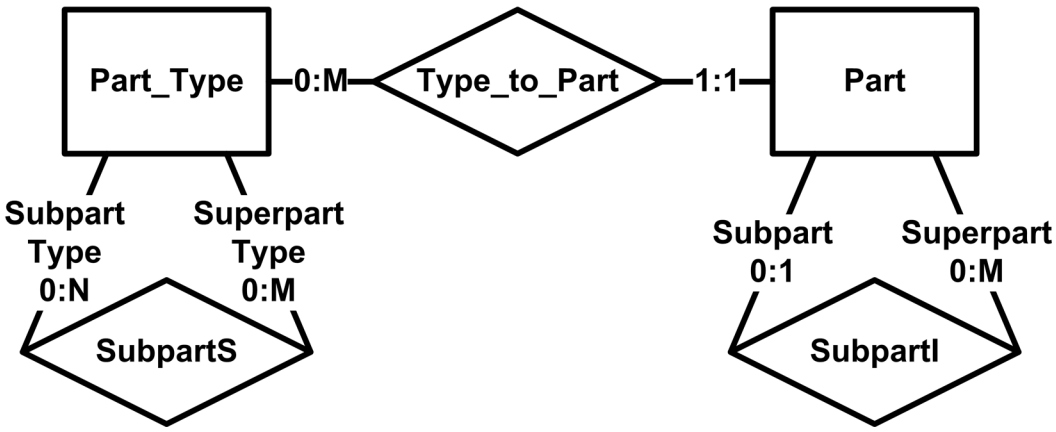

Figure 1.1: Hanyoda Motors Parts ER Diagram

A straightforward conversion of the ER diagram in Figure 1.1 to a relational design populated with some example values, results in the relations in Figures 1.2, 1.3, and 1.4. Note that the *SubpartI* relationship is subsumed in the *Part* table.

Using views to express constraints (one of two aspects of the duality we examine), we can specify two constraints that must hold in this scenario. The first of these constraints is a canonical constraint from the core relational model, a foreign key. We more formally introduce this constraint in Chapter 2; for now, we illustrate it by referring to Figures 1.3 and 1.4. In the case of both columns (*type_ID* and *child_type_ID*) in the *SubpartS* table, we naturally want to limit the values in both to the values that also appear in *type_ID* column in the *Part_Type* table. We formally specify these constraints using relational algebra in Examples 1.1.1 and 1.1.2.

Example 1.1.1 (type_ID Foreign Key Constraint)

part_ID	parent_part_ID	type_ID
T4655	NULL	T1
T4656	NULL	T2
E4888	T4655	E5
E1111	T4656	E3
B1234	E4888	B35
B5678	E1111	B35
C9876	T4655	C45
C9877	T4656	C32
⋮	⋮	⋮

Figure 1.2: Part Table

type_ID	description
T1	Full-Size Truck
T2	Compact Truck
E3	Turbo V8 Engine
E5	V8 Engine
E6	V6 Engine
B12	Small Block
B35	Big Block
C45	Extended Cab
C32	Cab
⋮	⋮

Figure 1.3: Part_Type Table

type_ID	child_type_ID
T1	E3
T1	E5
T2	E3
T2	E5
T2	E6
T1	C32
T1	C45
T2	C32
E3	B35
E5	B35
E6	B12
⋮	⋮

Figure 1.4: SubpartS Table

$\pi_{type\_ID}(SubpartS) \subseteq \pi_{type\_ID}(Part\_Type)$

Example 1.1.2 (child_type_ID Foreign Key Constraint)

$\pi_{child\_type\_ID}(SubpartS) \subseteq \pi_{type\_ID}(Part\_Type)$

Examples 1.1.1 and 1.1.2 are the formal descriptions of the foreign key constraints that we wish to enforce on the *SubpartS* table. Foreign keys are well-understood, and well-established techniques already exist for specifying them. However, other classes of constraints (such as the aforementioned business rules) do not benefit from similarly proven methods; another constraint from our example illustrates this deficiency. Specifically, we clearly do not want to allow the case where a part is related to itself via the relationships established in the Part table, yet no mechanism in the core relational model exists to preclude this case from occurring.

This example constraint provides a taste of the type of constraint that the view $\leftrightarrow$ constraint duality framework can easily express and we return frequently to the Hanyoda Motors scenario throughout the remainder of this dissertation. For now, we proceed to an overview of this document, including an outline of the hypotheses central to the research.

1.2 Overview of Research

The principal hypotheses of this dissertation are the following:

1. a view↔constraint duality exists in database systems and systems engineering domains and

2. the view↔constraint duality is a conceptually tractable and useful tool in the development of information systems.

In these hypotheses, duality refers to the ability of each to express the essence or meaning of the other. Although definitions of "view" and "constraint" vary by context, the fundamental premise of this research is the duality exists regardless of context. To provide tractability, the context for the research is first the database systems and systems engineering domains.

We begin in Chapter 2, where we discuss the necessary background information and introduce the notational conventions that we employ. Included in these topics are the Zachman Framework for Enterprise Architecture, the core relational model, relational algebra, relational calculus, Structured Query Language (SQL), and object-oriented analysis and design concepts.

We next expound on the view↔constraint duality in Chapter 3. In this exposition, we discuss the concepts foundational to our hypotheses and examine the difficulties that we encounter in establishing the duality. We also prove the duality's existence in a formal context (the core relational model) and then introduce additional contexts where we find empirical evidence for the duality.

To gather empirical evidence, it is necessary to probe these domains for instances of the duality, where the instances themselves are representative of the sub-domains in which they exist. To this end, we use the Zachman Framework for Enterprise Architecture to organize and guide in this exploration.

The Zachman Framework for Enterprise Architecture defines an architectural structure for organizing institutional artifacts and models; it is these artifacts and models which we examine to validate our hypotheses. The framework utilizes a grid structure with columns known as interrogatives and rows known as roles. The typical labels for the columns are the interrogatives: What, How, Where, Who, When, and Why. The rows commonly have role labels such as Specify, Design, and Build. Our hypotheses validation process tends to focus on the Specify and Design roles, as the Build role emphasizes the implementation-related representations that fall outside our purview. Please refer to Figure 2.1 for a graphical depiction of the framework.

The interrogatives are questions whose answers represent different abstractions of the enterprise. For example, the answers to the How question include a description of an enterprise's business processes. As for the roles represented by the rows, these are largely perspectives found throughout an enterprise; an example includes the owner stakeholder group naturally corresponding to the Owner role.

In the following, we move within the Zachman Framework from well-formalized areas to those more descriptive and intuitive in nature. Worth noting is that as we

make this shift, the first hypothesis becomes more difficult to formally express, and consequently, the validation itself becomes more descriptive. In general, the presence of strong formalisms provides a rich backdrop against which to validate the duality.

In Chapter 4, we begin the process of finding empirical evidence of the duality by closely inspecting the database systems domain (in the Zachman Framework, the What column). As the context in which we formally prove the duality is the core relational model and the core relational model is well entrenched as the paradigm *du jour* of the database systems domain, we establish in the previous chapter (Chapter 3) the general case of the duality in the database systems domain. In contrast, we focus in this chapter on specific instances of the duality in the database systems domain. We do, though, still have at our disposal the formalisms provided by the core relational model (namely relational algebra and relational calculus) as well as set containment and set equality. These formalisms enable us to express formally the instances that we address. We focus on the well-known constraints and dependencies found in the core relational model, utilizing relational algebra and set operations to illustrate the manner in which this facet of the duality holds. We also describe a set of extensions (including aggregate functions, transitive closure, and metadata operations) to the core relational model and investigate related classes of constraints. In these classes, we use views to express constraints that others have only been able to do descriptively.

In Chapter 5, we leave the data models and proceed to address models and methodologies at the conceptual level that are less formal and more descriptive. Here, we identify two archetypal elements that allow us to argue the existence of the duality in other areas of the Zachman Framework and thus to continue validating our hypotheses. The set of archetypal framework components consists of the process model and the time-based model. The process model resides in the How column and the time-based model in the When column. In both cases, we offer representations that allow us to see the manner in which the duality holds. We then continue our examination by delving into the duality in an area that we entitle cross-column modeling; we take this title from the manner in which modeling occurs across the columns of the Zachman Framework.

Having validated the first hypothesis, we focus on the second hypothesis. Our results from the first hypothesis grant us great flexibility and expressiveness, and we discuss the use of the duality as a tool in the development of information systems in Chapter 6.

From the inspection of the duality and its uses, it follows that the duality permits globally defined constraints that inhabit a sweet spot that is the confluence of formalism and practicality. Furthermore, the duality lends a credible definition of view that naturally flows across existing boundaries, granting a shared perspective on views and thereby enabling richer methodologies and models. We then conclude

with a summary of our results and a discussion of future work.

2

Background

This chapter provides background material and related work pertaining to the view-constraint duality. We start with the notational conventions that we employ in the remainder of the document. We then further investigate the Zachman Framework for Enterprise Architecture to help categorize contexts for views and constraints. We subsequently discuss the formal aspects of the relational model, as introduced by Codd in [Cod70]. We also provide an overview of query languages, including an introduction to relational algebra and calculus as well as the Structured Query Language (SQL). Next, we explore dependencies, constraints, and the SQL `VIEW` and `TRIGGER` mechanisms. We then conclude with a summary of the related work.

2.1 Notational Conventions

In this section, we list the notational conventions that we employ in the remainder of the document. Other conventions are introduced as necessary; such conventions have a more limited scope than the ones listed here.

- $R(A_1, \ldots, A_n)$ and $S(B_1, \ldots, B_m)$ are relation schemas.
- $\mathbf{r}$ is an instance of R, and $\mathbf{s}$ is an instance of S.
- $X, Y \subseteq \{A_1, \ldots, A_n\}$.
- $Z = \{A_1, \ldots, A_n\} - (X \cup Y)$.
- t, t_i, t_j, t_1, t_2, and t_3 denote tuples.
- $t_i.W = t_j.W$ denotes $\forall w \in W\ (t_i.w = t_j.w)$.
- $\forall i$, $X_i \subseteq \{A_1, \ldots, A_n\}$ and $\cup_{i=1}^{m} X_i = \{A_1, \ldots, A_n\}$.
- α denotes the transitive closure operation.
- $\circ$ denotes the relational composition operation.

2.2 Zachman Framework for Enterprise Architecture

In his IBM Systems Journal articles, John Zachman describes a Framework for Information Systems which defines an architectural structure for organizing institutional artifacts[Zac87, SZ92]. The expression of these artifacts reveals the interconnections and abstractions among structural elements and thereby facilitates the understanding of operational systems. In its latest form, "The Zachman Framework for Enterprise Architecture" presents an architectural model that is currently seeing more frequent use. The extensiveness and flexibility of the Zachman approach is demonstrated in a variety of applications from the design and construction of data warehouses[IZG97] to its role in academics as shown in the development of textbooks[WBD03].

The framework utilizes a grid structure with columns known as interrogatives and rows known as roles. The typical labels for the columns are the interrogatives: What, How, Where, Who, When, and Why. The rows commonly have role labels such as Specify, Design, and Build or equivalently Owner, Designer, and Builder. Please refer to Figure 2.1 for a graphical depiction of the framework.[1]

The interrogatives are questions whose answers represent different abstractions of

[1]This figure is courtesy of the Zachman Institute for Framework Advancement (ZIFA). It is not the most recent image from ZIFA, but it is the one that reproduces best in black and white.

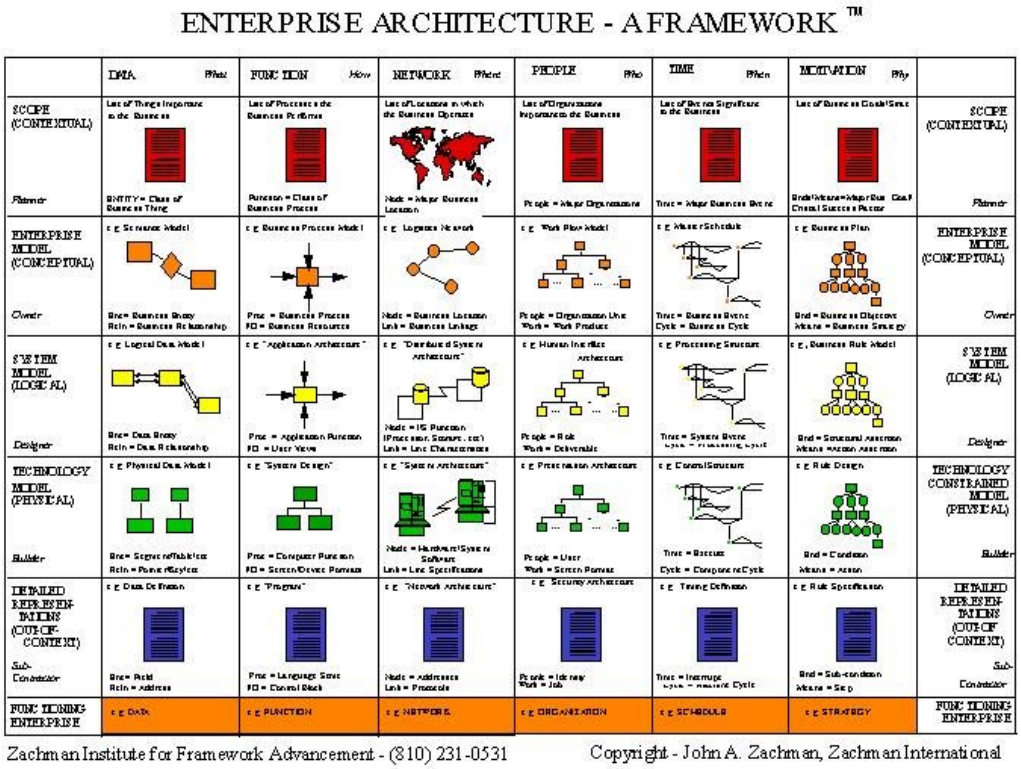

Figure 2.1: Zachman Framework for Enterprise Architecture

the enterprise. For example, the answers to the How question include a description of an enterprise's business processes. The roles represented by the rows are largely perspectives found throughout an enterprise; an example includes the owner stakeholder group naturally corresponding to the Owner role.

2.3 Core Relational Model

At the heart of the relational model is a relation. A relation R is a name and a finite set of attribute names $\{A_1, \ldots, A_n\}$, known as the "*schema*" or "*sort*" of R. We often write R to indicate only its name and $sort(R)$ to indicate $\{A_1, \ldots, A_n\}$. The *arity* of R is n.

A *tuple* t over R is a function mapping $sort(R)$ to the domain dom; if necessary, we distinguish the domains of the individual attributes as $dom(A_i)$. If $t(A_i)$ is not defined, we extend t to map A_i to `NULL`, for some distinguished symbol `NULL` not in dom. An *instance* of R, commonly denoted $\mathbf{r}$, is then a finite set of tuples over R.

Queries are another concept critical to the relational model. A query q is a mapping between relation instances. In the context of the relational model, the application of a query mapping to an input instance results in another relation instance. This extremely useful property is known as the property of relational closure.

2.4 Relational Query Languages

In this section, we discuss the relational query languages that we may use to write queries against the relational model. Please consult [AHV95] for additional information on the formal aspects of these query languages.

Relational Algebra

The relational algebra is a declarative query language, with queries defined by the composition of the following basic operations:

Selection: Written $\sigma_\theta(R)$, the selection operator selects tuples from its input relation R that satisfies the boolean condition θ. The θ expression is typically of the form $\alpha\phi\beta$ where α, $\beta \in sort(R) \cup dom(R)$ and ϕ is a binary relation often involving equality. θ may also be a boolean expression with base terms of the form $\alpha\phi\beta$ connected with the standard logical connectives.

Projection: Written $\pi_X(R)$ where $X \subseteq sort(R)$, the projection operator projects a subset of a relation's attributes, *i.e.*, X, and thus creates a new relation with attributes $sort(X)$.

Renaming: Written $\rho_Q(R)$ or $\rho_{\ldots,A\to B,\ldots}(R)$, the renaming operation allows the renaming of relations and individual attributes. In the case of $\rho_Q(R)$, ρ renames R to Q, while in the case of $\rho_{\ldots,A\to B,\ldots}(R)$, ρ renames the A attribute of the R relation to B.

Join: Written $R \bowtie_\theta S$ where θ is a boolean condition, the join operator is actually a Cartesian product operation followed by a selection operation, *i.e.*, $R \bowtie_\theta S = \sigma_\theta(R \times S)$, that results in a new relation with attributes $sort(R) \cup sort(S)$. In general, this only succeeds if $sort(R) \cap sort(S) = \emptyset$; renaming may be required

to achieve this. A special type of join operation known as a natural join, $R \bowtie S$, is a join that pairs tuples from R and S that agree on their common attributes, $sort(R) \cap sort(S)$.

Set operations: Written $\cup$, $\cap$, and $-$, respectively, the union, intersection, and set difference operations have their typical set semantics, with the added restriction that all relations serving as operands must have the same schema.

By convention, unmatched attributes in projections and renamings are ignored. That is, if $A \notin sort(R)$, then $\pi_{\ldots,A,\ldots}$ and $\rho_{\ldots,A \to B,\ldots}$ are interpreted as if the A and $A \to B$, respectively, were not present. In addition, to avoid concerns regarding operator precedence, we assume that relational algebra expressions are fully parenthesized.

Also worth noting is that relational algebra may be characterized as a procedural query language because it is possible to specify directly the evaluation tree used in answering the query. This is in contrast to "more" declarative languages such as relational calculus, which we discuss in the next subsection.

Relational Calculus

Relational calculus is based on first-order logic (*i.e.*, predicate calculus), and in fact, it is first-order logic using only finite models and without function symbols, as

Abiteboul, Hull, and Vianu discuss in [AHV95]. Because it is a logic, relational calculus is considered a declarative language; nevertheless, relational calculus and relational algebra do have equivalent expressiveness when the relational calculus formulas are *safe*. As Ullman discusses in [Ull88], the need for *safety* results from wanting to avoid interacting with infinite relations; for the reader interested in *safety*, we also suggest [AHV95].

Relational calculus comes in two flavors: tuple relational calculus and domain relational calculus. As Elmasri and Navathe discuss in [EN06], these two flavors differ in the types of variables that appear in their formulas. In the case of tuple relational calculus, the variables have as their domains sets of tuples (*i.e.*, relations), and thus each variable is assigned an individual tuple. On the other hand, in the case of domain relational calculus, each variable takes its value from the domain for the attribute that it represents. A more substantive difference is in quantification. That is, tuple calculus is typically safe by syntax and domain calculus by semantics, *i.e.*, by interpreting quantifiers only over the active domain. Nevertheless, tuple relational calculus and domain relational calculus have equivalent expressive power.

Structured Query Language (SQL)

The Structured Query Language (SQL) is the query language employed in most relational database management systems, and as Ullman discusses in [Ull88], SQL incorporates aspects of both relational algebra and relational calculus. In the following, we briefly describe the major components of interest; for a more detailed description, please consult the most recent ANSI/ISO SQL standards (SQL-92 (SQL2) [Ame92] and SQL-99 [Ame99]) or Connolly and Begg in [CB01].

As a query language, SQL manifests itself as `SELECT` statements. SQL also includes other data manipulation commands as well as aspects that concern data definition and data control. For our purposes, we focus on `SELECT` statements and the other data manipulation commands, *i.e.*, `INSERT`, `UPDATE`, and `DELETE` statements. We begin by examining the `INSERT`, `UPDATE`, and `DELETE` statements before addressing the `SELECT` statement.

The `INSERT`, `UPDATE`, and `DELETE` statements may operate on a single relation or possibly multiple relations. We limit our perspective in this document to the single relation case. As for the commands themselves, they perform the operations that their names suggested: `INSERT` inputs new tuples into relations, `UPDATE` modifies existing tuples, and `DELETE` removes tuples. One may consider the `UPDATE` statement as an atomic combination of a `DELETE` statement and then an `INSERT` statement with

implicit shared bindings. With each statement, it is possible to specify restrictive criteria in a `WHERE` clause that limits the influence of the command. As the `SELECT` statement may also contain a `WHERE` clause, we discuss the `WHERE` clause in the following.

As for the `SELECT` statement, its general form is the following:

`SELECT` $v_i.A_k$, ..., $v_j.A_l$

`FROM` R_i `AS` v_i, ..., R_j `AS` v_j

`WHERE` ϕ

where ϕ expresses a boolean condition and may incorporate the common logical connectives (`AND`, `OR`, and `NOT`) as well as comparative binary relations, such as $=$, $<$, $>$, *etc.* and the v_i, ..., v_j are a set of variable names and thus disjoint.

It is possible to express relational algebra and calculus queries in SQL, and we may express many but not all SQL queries in relational algebra and calculus. A well-known example of this difference in expressiveness is aggregation, as discussed by Libkin in [Lib01]. Henceforth, we denote this boundary as the core relational model, in that relational algebra and calculus can express the core relational queries; in Chapter 4, we revisit these limitations.

2.5 Constraints, Triggers, and Views

In this section, we delve into the relational model's dependencies as well as the constraints expressible by SQL. We also discuss the SQL TRIGGER and VIEW mechanisms.

Dependencies

In this subsection, we explain the four main dependencies found in relational databases. With each, we provide a brief description of the constraint and then give its formal definition. We explore these dependencies again in Chapter 4, and one may refer to [GMUW02] or [RG00] for additional information.

The first dependency is the functional dependency, and informally, a functional dependency occurs when values for a set of attributes Y depend directly on values for another set of attributes X. More formally, a relation instance $\mathbf{r}$ satisfies the functional dependency $X \rightarrow Y$ if $\forall t_1, t_2 \in \mathbf{r}$ $(t_1.X = t_2.X \Rightarrow t_1.Y = t_2.Y)$.

Another dependency from the relational model is the multivalued dependency. Given a set of fixed attribute values, a multivalued dependency holds when a second set of attribute values is entirely independent of a third set of attribute values. More precisely, a relation instance $\mathbf{r}$ satisfies the multivalued dependency $X \rightarrow\rightarrow Y|Z$ if $\forall$ $t_1, t_2 \in \mathbf{r}$ $(t_1.X = t_2.X \Rightarrow \exists\ t_3 \in \mathbf{r}$ $(t_1.XY = t_3.XY \wedge t_2.Z = t_3.Z))$.

The join dependency is a third relational model dependency, and a relation instance $\mathbf{r}$ satisfies the join dependency $\bowtie [X_1, \ldots, X_m]$ if $\mathbf{r} = (\bowtie_{i=1}^{m}(\pi_{X_i}(\mathbf{r})))$. Intuitively, a join dependency occurs when a relation is decomposed into multiple relations and the join of these relations results in a relation that equals the original relation. The decomposition is *lossless* in that no tuples were lost and no spurious tuples were generated as a result of the aforementioned join. It is worthwhile to note that functional dependencies and multivalued dependencies are special cases of the join dependency.

The final dependency is the inclusion dependency. In an inclusion dependency, a relation instance $\mathbf{r}$ satisfies the inclusion dependency $R[X] \subseteq S[Y]$ if $\pi_X(\mathbf{r}) \subseteq \pi_Y(\mathbf{s})$.

Each of these dependencies becomes a constraint on a schema R (or R and S for an inclusion dependency) if it is required that every instance $\mathbf{r}$ of R (respectively $\mathbf{r}$ and $\mathbf{s}$) satisfy that dependency.

Integrity Constraints

In this subsection, we briefly explore the constraints expressible by SQL. For a detailed discussion of integrity constraints, please see [GGGM98].

The first constraint that we discuss specifies a primary key. Intuitively, a primary key is an attribute or set of attributes whose values for a particular tuple are unique

across the entire relation. More formally, X is a primary key for $\mathbf{r}$ if $\forall\ t_1, t_2 \in \mathbf{r}$ $(t_1.X = t_2.X \Rightarrow t_1 = t_2)$.

Foreign keys are another type of integrity constraint. A foreign key is a restriction on an attribute from a relation that limits the values that the attribute may contain to the values found in a primary key attribute from another relation. Formally, $A_i \in sort(\mathbf{r})$ is a foreign key that refers to $B_j \in sort(\mathbf{s})$ if $\forall t_1 \in \mathbf{r}$, $\exists t_2 \in \mathbf{s}$ $(t_1.A_i = t_2.B_j)$.

Yet another type of integrity constraint is the value-based constraint. Constraints of this type serve to limit the values that an attribute may have. $A_i \in sort(\mathbf{r})$ satisfies the value-based constraint $\mathcal{C}$ if $\pi_{A_i}(\mathbf{r}) - \sigma_{\mathcal{C}}(\pi_{A_i}(\mathbf{r})) = \emptyset$.

Similar to the value-based constraints are tuple-based constraints. Instead of being defined on an attribute as is the case with value-based constraints, the tuple-based constraints are defined on relations. The relational instance $\mathbf{r}$ satisfies the tuple-based constraint $\mathcal{C}$ if $\mathbf{r} - \sigma_{\mathcal{C}}(\mathbf{r}) = \emptyset$.

The final type of integrity constraint that we discuss is an assertion. As Garcia-Molina, Ullman, and Widom indicate in [GMUW02], assertions are named boolean-valued expressions stated using SQL and are defined against entire database schemas as opposed to only attributes or relations. The view-based expressions that are the focus of this document have a natural implementation in the form of assertions; however, this aspect of the SQL standard is not widely implemented by the database management system vendors.

Views

An SQL `VIEW` is a named query that may be used as if it were a relation of the same name, *i.e.*, it may be queried and in some restricted cases, updated. A view – whether defined as an SQL `VIEW` or some other expression – may be implemented by recomputing the view as needed or as a precomputed and stored *materialized view*. There are also unnamed views, such as query expressions contained in `FROM` clauses or subqueries found in `INSERT`, `UPDATE`, and `DELETE` statements; naturally, an unnamed view cannot be queried as if it were a relation.

In addition, a distinction needs to be made between the view expression and the relation instance that results from the evaluation of the view expression. As "`VIEW`" may be used to describe both, we make the distinction where it is necessary. As a final consideration, the notion of *sort* is extended to SQL `VIEW` as the schema of the relation defined by the view expression, that is, the schema of the relation instance that results from the evaluation of the view expression.

Triggers

Relational Database Triggers are Event-Condition-Action (ECA) rules where an occurrence of an event causes the evaluation of a condition and if the condition is true, an action. An event is a data manipulation statement on a table (that

is, an INSERT, UPDATE, or DELETE) with an added temporal characterization (BEFORE, AFTER, or INSTEAD OF). As for the condition itself, any boolean-valued query is acceptable. Finally, the action contains the procedural steps to execute when the event is triggered and (if specified) the condition evaluates to true. It is the procedural nature of the action component that gives triggers their expressiveness.

Triggers are an extension of constraints, a powerful procedural alternative to the declarative nature of constraints. As discussed in [RG00], a common use of triggers is to enforce database consistency, a goal they obviously share with constraints; they are also often utilized to enforce complex business rules that constraints cannot capture. In fact, RDBMS allow the use of general purpose programming languages in the action clauses of their trigger definitions. For instance, Oracle allows PL/SQL in its action clauses; since PL/SQL is Turing-complete, this is indeed powerful but also dangerous. On a final note related to triggers, they typically fall into a more general area known as active databases.

2.6 Summary of Related Work

A well-understood property of SQL is that it may serve as a constraint specification language. Garcia-Molina, Ullman, and Widom discuss this in detail in [GMUW02]. Relational constraints and dependencies have been a frequent topic of research. Codd

introduced functional dependencies in [Cod72], while multivalued dependencies were studied individually by Fagin in [Fag77], Zaniolo in [Zan76], and Delobel in [Del78]. Very recently, Ceri, Di Giunta, and Lanzi in [CGL07] investigated the application of data mining techniques to discovering *pseudoconstraint* violations, while in [LS06], Lu and Scoggins explored the use of SQL extensions and the integration of satisfiability (SAT) solvers (into the RDBMS) in the solving of boolean constraint problems within a relational database.

Active databases are another area of substantial inquiry; this includes Ceri and Widom in [CW90] where they derive triggers to *repair* the data causing the violation of a constraint. Ceri and Widom continue their work on active database in [CW91] where they derive triggers based on SQL `VIEW`s to maintain incrementally the materializations of such views and in [CW94] where they derive triggers based on deductive database rules to maintain incrementally the deductive database's derived (*i.e.*, intensional) relations. Ceri carried on this work in [CFPT94] with Fraternali, Paraboschi, and Tanca, in which they derive triggers from integrity constraints to enforce the constraints. Additional work in the area of constraint maintenance has been performed by Casanova, Tucherman, and Furtado in [CTF88] and Morgenstern in [Mor83].

As for relational views, they too have received significant attention, especially as they concern maintenance (that is, the maintenance of materialized views when the

base relations are updated) and update (that is, the translation of updates to views into updates to base relations). Buneman and Clemons present pioneering work on view maintenance in [BC79]. Other authors with work on view maintenance include Koenig and Paige in [KP81]; Paige again in [Pai82]; Blakeley, Larson and Tompa in [BLT86]; Gupta, Katiyar, and Mumick in [GKM92]; and Qian and Wiederhold in [QW91]. One well-known property of view maintenance concerns its connection to incremental maintenance of integrity constraints. In fact, this connection bolsters our claims concerning the view-constraint duality, especially in the relational database context; we delve into this in Chapter 3. Several authors discuss this connection, including Bernstein, Blaustein, and Clarke in [BBC80], Henschen, McCune, and Naqvi in [HMN82], and Ross, Srivastava, and Sudarshan in [RSS96].

As for the view update problem, it too has been the recipient of extensive research. In [FC85], Furtado and Casanova provide a survey of this research. Furtado, Sevcik, and dos Santos in [FSdS79], Rowe and Shoens in [RS79], Dayal and Bernstein in [DB82], and more recently, Lechtenbörger and Vossen in [LV03] all provide pioneering work in this area.

Constraints and views have also been a topic of research in the object-oriented analysis, design, and implementation areas, and several standards organization have addressed them in their organizations' publications. Concerning views, Albin in [Alb03] discusses them in the context of software architectures, while in [Wre98],

Wrembel examines the notion of object views from an object-oriented perspective. Significant research has been conducted in the area of the relationship between objects and relational databases; this includes Keller in [Kel86] and several textbooks [AHV95, Ull88, GMUW02, RG00, CB01, EN06]. In addition, Roberts, Berry, Isensee, and Mullaly discuss a use of the view notion in the design of user interfaces in [RBIM98].

As for constraints, a widely used set of object-oriented modeling techniques is the Unified Modeling Language (UML); see [FS00, PJ00] for additional information. An extension to UML for specifying constraints on UML models is the Object Constraint Language (OCL). Two useful OCL references are the OCL 2.0 specification [OMG05] and the book authored by Warmer and Kleppe [WK03]. As Beckert and Trentelman examine in [BT05], OCL does allow recursion and can express properties of relations (*e.g.*, transitive closure) that first-order logic cannot.

As we mention earlier, standards organizations have given significant attention to views and constraints in their publications, albeit as independent concepts. For instance, the Institute of Electrical and Electronics Engineers (IEEE) have formally defined views in the software engineering context [IEE00]. In addition, in [ISO00], the International Organization for Standardization (ISO) addresses the relationship between views and constraints by acknowledging the role that views play in understanding models and their constraints.

3

The View↔Constraint Duality

In this chapter, we expound on the view↔constraint duality and the hypotheses that are the foci of this research. In this exposition, we discuss the concepts foundational to our hypotheses by way of an examination of the difficulties that we encounter in establishing the duality. We also begin the process of establishing the duality by proving its existence in a formal context and then introducing the informal situations that provide additional contexts for our efforts. We conclude this chapter with a preliminary discussion of the duality as a tool in the development of information systems.

We start with a restatement of our two hypotheses:

1. **a view↔constraint duality exists in database systems and systems engineering domains and**

2. **the view↔constraint duality is a conceptually tractable and useful tool in the development of information systems.**

Duality refers to the ability of each to express the essence or meaning of the other. Although the methods employed to express views and constraints vary by context, the fundamental premise of this research is that this duality exists regardless of context. To provide tractability, the context for the research is the database systems and systems engineering domains.

3.1 Challenges in Establishing the Duality

Before we address the specific challenges we face, it's necessary to note that although the duality holds equally well in each direction (as we see in the following), constraints expressed by views are commonly more useful. Intuitively, this stems from a couple of factors. The first of these factors concerns the richer data types that are present in views. These richer types permit more expressive statements, and we address this in the next subsection. As for the second factor, it concerns the indirect role that constraints can play in expressing views, and we postpone discussing this indirection until we establish the duality in a formal context.

Another point of emphasis concerns an aspect of the nature of constraints. More specifically, constraints specified at the model level constrain instances. For example,

one specifies primary keys on relational table schemas (*i.e.*, the relational model) that the relational database management system enforces on the instances of the tables. Furthermore, although it is less commonly done, this characteristic extends to the meta-model level; that is, one specifies constraints on the meta-model that constrain models based on the meta-model. As we investigate instances of the duality, we observe this aspect of the nature of constraints.

Having discussed these general items, we now examine the aforementioned challenges:

1. the type mismatch between views and constraints, and

2. the degrees of formality in the contexts we investigate.

The remainder of this section concerns these challenges.

Type Mismatch

The type mismatch between views and constraints creates an impediment to the formation of a precise statement of the first hypothesis, as views and constraints are considered to be very distinct concepts. Constraints are either true (satisfied) or false (not satisfied). For example, in the core relational model, constraints can be expressed as closed first-order logic formulae that naturally evaluate to boolean

values. On the other hand, as we briefly discussed in Chapter 1, views are an organized subset of a whole, where the subset is chosen by some criteria; both the "whole" and the "organized subset" vary according to context. In the core relational model, this whole is a database schema, and both relational algebra and relational calculus can express views on said schema.[1] Due to the property of relational closure, the resulting organized subset is a relation. In general, bridging the gap created by this type mismatch is critical to an effective characterization of the duality.

Throughout this research, we adhere to a set-theoretic approach for defining the underlying whole, and thus, any language that we utilize to express views performs its operations against sets. As a result of using set theory, we inherit the set operations that we discuss in Chapter 2 (that is, union, intersection, and set difference) as well as set containment and set equality. Set containment and set equality prove to be critical in enabling us to use views to express constraints. By embedding view expressions (which evaluate to sets) into expressions that utilize set containment and set equality, we construct truth-valued expressions that are indeed constraints, and thereby, we address the aspect of the mismatch that pertains to the expression of constraints using views.

On the other hand, to address the aspect of the mismatch that concerns the expression of views using constraints, we need a mechanism that utilizes logical expressions

[1] Worth noting is that the languages for expressing views also vary according to context.

to create sets. Thankfully, such a mechanism exists and is well-known – the notion of set comprehension. Definition 3.1.1 contains a general form of set comprehension; see [AU94] for additional information on set comprehension.[2]

Definition 3.1.1 (Set Comprehension)

Given a boolean predicate or property, P, the set comprehension of P, with respect to some domain $\mathcal{D}$, is $\{x \mid x \in \mathcal{D} \text{ and } P(x)\}$.

This simple idea is in some sense too powerful, leading to Russell's paradox if unrestricted set quantification is allowed in P. This further motivates our desire to limit the expressive power of languages.

As we saw in Section 2.4 pertaining to relational calculus, the concept of set comprehension plays a major role in the expression of relational calculus queries, and it is also critical in allowing us to use constraints to express views. More specifically, if we cast a constraint as the property P in Definition 3.1.1 and some underlying universe as the domain $\mathcal{D}$, then the resulting set is a view of that underlying universe. Hence, through the notion of set comprehension, we are able to construct views by utilizing constraints.

[2]The authors denote this as the definition of sets by abstraction.

Degrees of Formality

Having employed solutions to remedy both aspects of the type mismatch, we now deal with the difficulty stemming from the widely varying degrees of formality in the contexts in which views and constraints occur. Whereas the type mismatch issue concerned the precise stating of the first hypothesis, the difficulty arising from the degrees of formality is chiefly an issue that pertains to demonstrating that the first hypothesis is indeed true. As we stated earlier, the context for the research is first the database systems and systems engineering domains, and it is in these areas that we demonstrate the validity of the first hypothesis.

As we proceed through this validation process and move from database systems to the systems engineering domain, the first hypothesis becomes more difficult to express formally, and consequently, the validation itself becomes more descriptive. In general, the presence of strong formalisms provides a rich backdrop against which to validate the duality. However, this should not be construed as a weakness of the duality; instead, it is a strong indicator of the pervasiveness of the duality, in that the duality holds in both the highly formal cases as well as the informal and descriptive cases.

3.2 Formal Context

Our preceding discussion of set containment and set equality as well as set comprehension leads us to an outline of an approach for verifying the duality:

Case 1: Constraints expressed by Views

Construct constraints by the use of set containment and set equality in view expressions, where the view expressions naturally evaluate to sets.

Case 2: Views expressed by Constraints

Construct views by the use of set comprehension, where the properties in the set comprehension statements are constraints.

Building on this approach, Theorem 3.2.1 proves the duality in the formal context of first-order logic. It is extremely important to note that references to first-order logic throughout this dissertation concern the specialization of first-order logic found in database theory, that is, the version that omits functions, includes equality, and pertains to finite models. Furthermore, it is equally important to emphasize that since views and constraints can be expressed formally in first-order logic (FOL), we may state and prove the first hypothesis as a formal theorem. In addition, we assume in the following that the views are not inherently empty; that is, we assume that for each view expression containing a relation R, there exists an instance $\mathbf{r}$ of R such that the resulting view is not empty when the view expression is evaluated using $\mathbf{r}$.

For example, we disallow a view that specifies an integer value a is greater than itself ($a > a$) as a part of its condition. We extend this assumption to view expressions that contain two or more different relations; in these situations, we assume a set of relational instances that generate a non-empty view.

Theorem 3.2.1 (View↔Constraint Duality in First-Order Logic)

The view↔constraint duality exists in first-order logic.

Proof:

Case 1: Constraints expressed by Views

A constraint is a closed FOL formula Φ.

Case 1.1: Let $\Phi = \exists x\ (\Phi'(x))$.

Define view $V = \{x \mid \Phi'(x)\}$, where $x \in \mathcal{D}$.

The expression of the constraint Φ is $V \neq \emptyset$.

Case 1.2: Let $\Phi = \forall x\ (\Phi'(x))$.

Define view $V = \{x \mid \Phi'(x)\}$, where $x \in \mathcal{D}$.

The expression of the constraint Φ is $V = \mathcal{D}$.

Case 2: Views expressed by Constraints

Let a view $U = \{y \mid \Xi(y)\}$, where $\Xi(y)$ is a FOL formula that has no free variables with the exception of possibly y.

The associated constraints are $\exists y\ (\Xi(y))$ and $\forall y\ (\Xi(y))$. □

Case 1 of Theorem 3.2.1 exhibits clearly the capacity of set containment/equality to express the quantification performed in the closed FOL formulae, *i.e.*, constraints. For instance, the existential quantifier(s) ($\exists$) in a formula may be expressed by stating that a view-based expression is not equal to the empty set ($\emptyset$). Moreover, in cases involving constraints that are expressed by stating that a view-based expression is equal to the empty set (in general, $V = \emptyset$ where V is a view-based expression), the view-based expressions in isolation (that is, without the parts of the statements that specify equality with the empty set) indicate the violators of the constraints; we see instances of this observation in the subsequent chapters.

In addition, in case 2 of the theorem, we see an instance of the notion that we mentioned earlier concerning the indirect role that constraints can have in the expression of views. By taking a constraint and removing its quantifier, we create an expression that, when augmented with set comprehension, is a relational calculus query by definition, and since relational calculus queries can express views in the core relational model, we can characterize the expression as a view. Consequently, we can utilize constraints to express views in the core relational model. However, this utilization is not direct; it's the result of trivially changing the closed formula to an open one.

First-order logic offers us a well-accepted formalism for demonstrating that the duality holds. Other, less formal settings for the duality also exist, and in the next section, we explore our methods for spanning this wide spectrum of contexts.

3.3 Informal Contexts

As we stated in Chapter 1, we use the Zachman Framework for Enterprise Architecture to organize and guide our activities. By using the framework, we effectively characterize the space (that is, the wide range of contexts) over which our activities range, and consequently, we reduce the difficulty of verifying the duality across this space to demonstrating that the duality holds in the representative cases of the framework. At first glance, extrapolating from the representative cases to the entire population may seem faulty. However, we advance the notion that in spirit the representative cases capture the essence of their peers, and thus while the syntax may differ, the semantics have commonality. As a result, we have an approach that is thorough yet manageable.

It is necessary to state explicitly that in these discussions we see two different kinds of validation. In the case of FOL, the validation is a simple formal proof of the duality. On the other hand, with the use of the Zachman Framework that spans a wide range of contexts, we endeavor to observe empirical evidence of the duality – that is, occurrences where the concepts of view and constraint appear together and exhibit the duality – and thus validate empirically the duality.

3.4 Uses of Duality

We now turn our attention briefly to the second hypothesis. As the hypothesis states, the duality is a conceptually tractable and useful tool in the development of information systems. As we discuss in Chapter 6, the use of the duality as a tool mainly involves the use of views to constrain. Furthermore, the tool is conceptually tractable in that it utilizes a familiar medium, views, and useful in that it concerns practical problems.

As was the case with the first hypothesis, the second hypothesis finds its most natural contexts for validation in the areas that are most formal. For example, the relational database is a prime target for demonstrating that the duality has uses with the virtues we listed.

4

Duality in Data Models

In this chapter, we present empirical evidence for the first hypothesis in the database systems domain (in the Zachman Framework, the What column). In Theorem 3.2.1, we proved the existence of the duality in the core relational model. Here, we complement that formal result with empirical results for the well-known constraints and dependencies found in the core relational model.

We concentrate first on the canonical dependencies and then discuss the well-accepted constraints. For each constraint or dependency, we present first the classic definition and then provide a definition that utilizes views to express the constraint or dependency.

Additionally in this chapter, we discuss relational algebra extensions that permit the expression of constraints that involve metadata, transitive closure, and aggregation. These extensions yield a language capable of representing these types of

constraints via views.

4.1 Relational Dependencies

We begin with the classic dependencies from the core relational model, as they serve as the basis for the constraints in the next section. We use the conventions specified in Chapter 2.

Functional Dependency

The first dependency we investigate is the functional dependency. Its definition appears in Definition 4.1.1, and the definition using views to express a general functional dependency is in Definition 4.1.2.

Definition 4.1.1 (Functional Dependency)

$\mathbf{r}$ satisfies the functional dependency $X \rightarrow Y$ if the following holds:

$$\forall t_1, t_2 \in \mathbf{r}\ (t_1.X = t_2.X \Rightarrow t_1.Y = t_2.Y)$$

Definition 4.1.2 (Functional Dependency Using Views)

$$\sigma_{\bigvee_{y \in Y} \mathbf{r}.y \neq \mathbf{s}.y}(\mathbf{r} \bowtie_{\bigwedge_{x \in X} \mathbf{r}.x = \mathbf{s}.x} \rho_{\mathbf{s}}(\mathbf{r})) = \emptyset$$

Multivalued Dependency

Having discussed functional dependencies, we now move to multivalued dependencies. A definition for them appears in Definition 4.1.3, and the corresponding definition using views to express the general case of a multivalued dependency is in Definition 4.1.4.

Definition 4.1.3 (Multivalued Dependency)

$\mathbf{r}$ satisfies the multivalued dependency $X \rightarrow\rightarrow Y|Z$ if the following holds:

$$\forall t_1, t_2 \in \mathbf{r}\ (t_1.X = t_2.X \Rightarrow \exists t_3 \in \mathbf{r}\ (t_1.XY = t_3.XY \wedge t_2.Z = t_3.Z))$$

Definition 4.1.4 (Multivalued Dependency Using Views)

$$\mathbf{r} = (\pi_{XY}(\mathbf{r}) \bowtie \pi_{XZ}(\mathbf{r}))$$

Definition 4.1.4 is already used in explanations of multivalued dependencies, providing evidence that the first hypothesis has been used in specific cases.

Join Dependency

Having discussed the functional and multivalued dependencies, next we address the join dependency. We define it in Definition 4.1.5 and observe that this definition is in fact view-based.

Definition 4.1.5 (Join Dependency)

r satisfies the join dependency $\bowtie [X_1, \ldots, X_m]$ if the following holds:

$\mathbf{r} = (\bowtie_{i=1}^{m} (\pi_{X_i}(\mathbf{r})))$

That the join dependency is most naturally stated using views lends additional credibility to that part of the duality that we are addressing in this section, namely using views to express constraints.

Inclusion Dependency

We conclude our look at dependencies with the inclusion dependency. Definition 4.1.6 contains the definition for the inclusion dependency; we observe that this definition is also view-based.

Definition 4.1.6 (Inclusion Dependency)

r satisfies the inclusion dependency $R[X] \subseteq S[Y]$ if the following holds:

$\pi_X(\mathbf{r}) \subseteq \pi_Y(\mathbf{s})$

The inclusion dependency's view-based definition stems from stating this dependency using the subset notation. Taken in conjunction with the join dependency, we see that the duality not only offers the ability to use views to express the relational model's classic dependencies but also provides the mechanism for most naturally expressing some of them.

4.2 Relational Constraints

We turn our attention to the canonical constraints from the relational model. As we did with the dependencies, we present the definition before stating an expression using views that captures the essence of the constraint.

Primary Key

The first constraint that we encounter is the primary key. Its definition appears in Definition 4.2.1, and Definition 4.2.2 uses views to express the primary key in a general manner that follows from the definition.

Definition 4.2.1 (Primary Key)

X is a primary key for $\mathbf{r}$ if the following holds:

$\forall t_1, t_2 \in \mathbf{r}\ (t_1.X = t_2.X \Rightarrow t_1 = t_2)$

Definition 4.2.2 (Primary Key Using Views)

$\sigma_{\bigvee_{y \in sort(\mathbf{r})} \mathbf{r}.y \neq \mathbf{s}.y}(\mathbf{r} \bowtie_{\bigwedge_{x \in X} \mathbf{r}.x = \mathbf{s}.x} \rho_{\mathbf{s}}(\mathbf{r})) = \emptyset$

Foreign Key

Next, we investigate the foreign key. Definition 4.2.3 states the definition for the foreign key, and its view-based expression appears in Definition 4.2.4.

Definition 4.2.3 (Foreign Key)

A_i in $\mathbf{r}$ is a foreign key that refers to B_j in $\mathbf{s}$ if the following holds:

$$\forall t_1 \in \mathbf{r}, \exists t_2 \in \mathbf{s}\ (t_1.A_i = t_2.B_j)$$

Definition 4.2.4 (Foreign Key Using Views)

$$\pi_{A_i}(\mathbf{r}) \subseteq \pi_{B_j}(\mathbf{s})$$

Value-based Constraint

Having inspected primary and foreign keys, we proceed to a definition of value-based constraints, as contained in Definition 4.2.5. As for the view-based expression of value-based constraints, this occurs in Definition 4.2.6.

Definition 4.2.5 (Value-based Constraint)

A_i in $\mathbf{r}$ satisfies the value-based constraint $\mathcal{C}$ if the following holds:

$$\forall t \in \mathbf{r}\ [\mathcal{C}(t.A_i)]$$

Definition 4.2.6 (Value-based Constraint Using Views)

$$\pi_{A_i}(\mathbf{r}) - \sigma_{\mathcal{C}}(\pi_{A_i}(\mathbf{r})) = \emptyset$$

Tuple-based Constraint

We complete our look at constraints by tackling tuple-based constraints. We define tuple-based constraints in Definition 4.2.7 and then state its view-based expression

in Definition 4.2.8.

Definition 4.2.7 (Tuple-based Constraint)

r satisfies the tuple-based constraint $\mathcal{C}$ if the following holds:

$\forall t \in \mathbf{r}\ [\mathcal{C}(t)]$

Definition 4.2.8 (Tuple-based Constraint Using Views)

$\mathbf{r} - \sigma_{\mathcal{C}}(\mathbf{r}) = \emptyset$

Notice that we only address in this chapter that aspect of the duality that concerns constraints expressed by views. This results from the absence of a set of canonical set of views in the core relational model. Given that views are a general notion in the core relational model (that is, any relational algebra expression can represent a view[1]), the notion of a "canonical" view does not apply, and consequently, we omit material pertaining to empirical evidence for views expressed by constraints.

4.3 Relational Extensions

Having previously discussed the duality of views and constraints in the relational database context, it is natural to demonstrate instances in that context where the duality enables facile constraint specifications that otherwise either require a prose description or only find representation in the implementation. In the sequel, we discuss

[1] The same holds for any relational calculus query.

the concepts of metadata, transitive closure, and aggregation; for each, we describe an operation that implements the concept in the relational database context. As we see in the following, constraints in this context not only include those enumerated earlier in this discussion but also encompass those that have a more semantic flavor, such as account balance restrictions and cardinality constraints. Also contained in these classes of constraints are recursively defined constraints, such as bill of material constraints, and constraints involving restrictions pertaining to metadata.

These constraints include concepts not expressible by first-order logic; consequently, to satisfy them, we need to extend our relational faculties (that is, relational algebra and calculus) to permit the expression of constraints that involve metadata, transitive closure, and aggregation and thereby, to yield a language capable of representing these constraints via views. This makes an erstwhile covert theme more overt; namely, that to suggest views as a means to express constraints, it is necessary to introduce a language with sufficient power to express the constraints without falling into the trap of allowing the language to be Turing-complete. For now, let us segue to metadata constraints.

Metadata Constraints

We introduce metadata constraints by way of an example from the Hanyoda Motors scenario in Chapter 1. Hanyoda Motors relies on several suppliers to provide

parts for its truck assembly process, and these suppliers store their part information in many different database schemas. As part of its efforts to closely integrate its supply chain, Hanyoda offers its suppliers the option to communicate their inventory information electronically through such means as EDI or ebXML (Electronic Business XML). For its largest suppliers, Hanyoda even allows the suppliers to integrate directly into Hanyoda's inventory management information system through the use of supplier-specific *Part* tables that in essence serve as the suppliers' interface into the system.

To enable these suppliers to more easily integrate their own systems with the Hanyoda inventory management system, Hanyoda does give them some autonomy in determining the schemas for their *Part* tables; however, the suppliers must meet certain schema requirements, such as column names. As a result, Hanyoda does not have to build a custom interface to each table and instead can retrieve information from the suppliers' *Part* tables in a standard fashion.

Two such suppliers, Bloomington Metalworks and Muskegon Tool, have vastly different table schemas for storing such data, but as we just indicated, Hanyoda gains uniformity by enforcing Examples 4.3.2 and 4.3.3 on these suppliers' *Part* table[2] schemas.

[2]Views may be used in place of tables to ensure uniformity. Using this approach, the suppliers would create views based on their *Part* tables, and these views' schemas would match Hanyoda's *Part* table schema. The constraints would then involve the views instead of the tables.

In Examples 4.3.2 and 4.3.3, we utilize the δ operation from Definition 4.3.1. In [WR05], Wyss and Robertson present a version of the δ operation for federated databases. We provide the definition before proceeding to the examples.

Definition 4.3.1 (δ Operation)

Let $R(A_1, A_2, \ldots, A_n)$ be a relation schema. The δ operation on R, $\delta(R)$, is simply $sort(R)$; that is, $\delta(R) = \{A_1, A_2, \ldots, A_n\}$.

Example 4.3.2 (Bloomington Metalworks Schema Constraint)

$\delta(Part) \subseteq \delta(BloomingtonMetalworksPart)$

Example 4.3.3 (Muskegon Tool Schema Constraint)

$\delta(Part) \subseteq \delta(MuskegonToolPart)$

As these examples illustrate, the capacity to specify constraints on the metadata via views on the metadata is a powerful tool and exemplifies the value gained by employing the duality. Furthermore, we utilize this example to illustrate the ability of the "views expressing constraints" approach to easily indicate constraint violators in a natural way. To elucidate this point, we reformulate Example 4.3.2 into Example 4.3.4.

Example 4.3.4 (View Based on Bloomington Metalworks Schema Constraint)

$\delta(Part) - \delta(BloomingtonMetalworksPart)$

The expression in Example 4.3.4 pinpoints any column name in the *Part* table that does not have a matching column name in the *BloomingtonMetalworksPart* table. In some sense, Example 4.3.4 is an evolution of Example 4.3.4, in that Example 4.3.4 both enforces the constraint (by employing $\emptyset$ as true) and indicates violators. This is clearly an advantage of the declarative approach employed by the duality; a procedural method not only has to determine if the constraint has been violated but also must "remember" the instances that violated the constraint when such violations do occur.

We conclude this discussion of metadata constraints by returning to our original examples concerning metadata. Figure 4.1 depicts graphically the constraints specified in Examples 4.3.2 and 4.3.3 by employing extensions to the standard ER notation. These extensions include the following:

- encircled entities and relationships that participate as operands in operations;
- encircled operators whose operands are either entities, relationships, or other operators; and
- directed edges that indicate the participation of entities, relationships, or operators in an operation.

Please note that the $\subseteq$ operation naturally requires an ordering on its operands, unlike union and intersection for instance. To account for this, the graphical notation

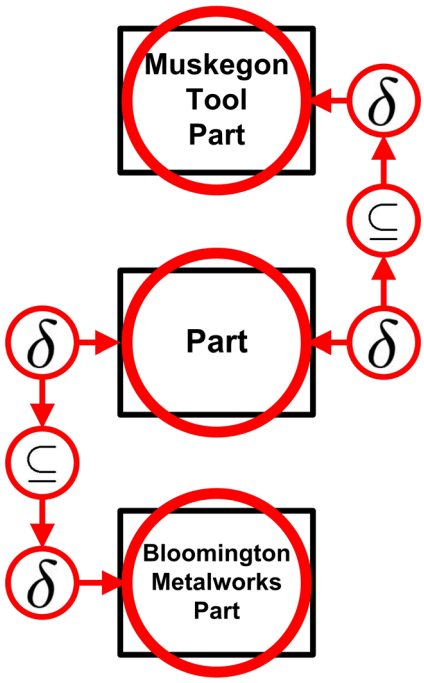

Figure 4.1: ER Diagram with Metadata Constraints

using $\subseteq$ must have a directed edge pointing to the node housing the $\subseteq$ as well as a directed edge pointing from that node.

As Figure 4.1 illustrates, introducing notation for the metadata extension into the standard ER notation permits the easy specification of metadata constraints in the ER diagram. Moreover, by incorporating the metadata extension into the ER notation via the aforementioned notational additions, we enable the modeler to specify metadata constraints naturally as a part of the data modeling process, and consequently, we satisfy the objective of minimizing constraint specification displacement.

Transitive Closure Constraints

Having illustrated the manner in which views can express metadata constraints, we now turn our attention to another class of constraints, those involving transitive closure. As we did with the metadata constraints, we initiate our discussion of transitive closure constraints with an example. Consider the Hanyoda Motors Parts entity-relationship diagram in Figure 1.1 from Chapter 1. As we discussed in that chapter, we clearly do not want to allow the case where a part is related to itself via the relationships established in the *Part* table, yet no mechanism in the core relational model exists to preclude this case from occurring.

A convenient method for enforcing this constraint is to use transitive closure. More specifically, taking the transitive closure of the relation created by projecting the *parent_part_ID* and *part_ID* from the *Part* table (ignoring the tuples where the *parent_part_ID* is not specified) and then selecting from the resulting relation (that is, the one resulting from the transitive closure operation) any tuple whose *parent_part_ID* equals its *part_ID*. If the relation that results from the selection is not empty, then the constraint has been violated. A more concise representation of the constraint appears in Example 4.3.5.

Example 4.3.5 (Acyclic Constraint)

$\sigma_{parent\_part\_ID=part\_ID}(\alpha(\pi_{parent\_part\_ID,part\_ID}(Part))) = \emptyset$

This example highlights the critical need for transitive closure to be available to views that express constraints. In [Imm87], Immerman introduced Transitive Closure Logic as an extension of First-Order Logic (FO) that includes the ability to define transitive closure; FO + TC denotes Transitive Closure Logic. To First-Order Logic, it adds an operator α that defines the transitive closure of a particular FO formula φ. More formally, Definition 4.3.6 contains the specification of the syntax.

Definition 4.3.6 (α Operator Syntax)

Given a First-Order formula φ with the free variables $\vec{x} = x_1, \ldots, x_n$ and $\vec{y} = y_1, \ldots, y_n$, as well as two n-tuples of terms, $\vec{s} = s_1, \ldots, s_n$ and $\vec{t} = t_1, \ldots, t_n$. Then $[\alpha_{\vec{x},\vec{y}}\ \varphi(\vec{x},\vec{y})](\vec{s},\vec{t})$ is a formula, where all free occurrences of $\vec{x}$ and $\vec{y}$ in φ are now bound by the α operator.

As for the semantics of the α operator, Definition 4.3.7 contains a detailed explanation of them.

Definition 4.3.7 (α Operator Semantics)

Given a relation[3] R satisfying φ with arity 2n. Furthermore, let S and T denote relations (both also with arity 2n) and ($\vec{x}$,$\vec{y}$), ($\vec{y}$,$\vec{z}$), and ($\vec{x}$,$\vec{z}$) denote 2n-tuples, where $\vec{x} = x_1, \ldots, x_n$, $\vec{y} = y_1, \ldots, y_n$, and $\vec{z} = z_1, \ldots, z_n$. Also, interpret the operator := as assignment, and construe the comparison of $\vec{x}$, $\vec{y}$, and $\vec{z}$ for equality (*e.g.*, $\vec{x} \neq \vec{y}$)

[3] Or more accurately a relational instance, to emphasize this refers to actual tuples.

to be the pairwise comparison of elements, such as $x_i = y_i$ where $1 \leq \text{i} \leq \text{n}$. The semantics of the α operator are as follows:

S := R

T := $\emptyset$

While S $\neq$ T do:

T := S

For each $(\vec{x},\vec{y}) \in$ S do:

If $\vec{x} \neq \vec{y}$ and $\exists\vec{z}$ such that $(\vec{y},\vec{z}) \in$ S, add $(\vec{x},\vec{z})$ to S

Return T

Having introduced the α operator, we can now proceed to a discussion of its formal properties. As Grädel and McColm demonstrate in [GM96], FO + TC is equivalent to stratified linear Datalog, and the data complexity of both is in QNLOGSPACE. Thus, FO + TC offers a language that provides the necessary expressiveness in a safe manner. Also, worth noting is that we continue to operate in the domain of relational databases, since the α operator returns a relation.

Please note that, although we did previously introduce the α operator using a logic-theoretic (namely FO + TC), it is an easy transition to formulate it algebraically, because of the relationship between Relational Calculus and Relational Algebra. Because of their equivalence, Relational Algebra extended with the α operator is equivalent to FO + TC. Henceforth, we largely prosecute the discussion in

algebraic terms.

Now that we have discussed the pertinent formal properties that hold for our extension, it is beneficial to mention some instances of constraints that utilize transitive closure. Examples include constraints on bill of materials such as the one we described earlier. In addition, a natural set of constraints that incorporate transitive closure are those involving organizational hierarchies, such as limitations on an organization's structure. For instance, one may wish to prevent a cyclic chain of command from occurring. A similar situation may apply to another group of constraints requiring transitive closure, genealogy. Here, one may again have the need to enforce acyclicity, in this case to prevent a person from being his or her own ancestor (or descendant, depending on perspective).

We conclude this discussion of transitive closure constraints by returning to our original example. Figure 4.2 depicts graphically the constraint specified in Example 4.3.5 by employing extensions to the standard ER notation. These extensions include a looping arrow to represent α and a dashed line (originating from the looping arrow) with an encircled $\neq$ operator comparing *parent_part_ID* and *part_ID*; this dashed line arrangement represents the restriction that permits only cases where *parent_part_ID* and *part_ID* are not equal.

As Figure 4.2 illustrates, introducing notation for the transitive closure extension into the standard ER notation mirrors the benefit that the metadata extension

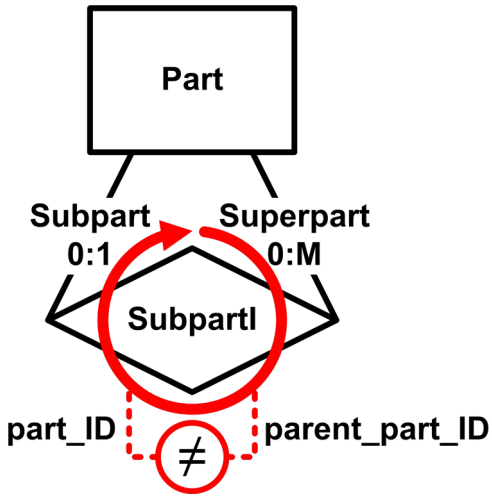

Figure 4.2: ER Diagram with Extensions

provided, namely the easy specification of transitive closure constraints in the ER diagram. Moreover, just as it did with the metadata extension, incorporating the transitive closure extension enables the modeler to specify transitive closure constraints naturally as a part of the data modeling process, again satisfying the objective of minimizing constraint specification displacement.

Aggregation-related Constraints

Having seen metadata and transitive closure constraints, we now turn our attention to constraints involving aggregation. As we discuss earlier, constraints pertaining to aggregation include account balance restrictions and cardinality constraints, and as is the case with metadata and transitive closure constraints, we need an extension

to relational algebra to permit the expression of aggregation-related constraints. Just such an operation, the grouping or γ operation, exists; Garcia-Molina, Ullman, and Widom discuss this operator in [GMUW02], with its introduction coming in [GHQ95]. We define the γ operator in Definition 4.3.8.

Definition 4.3.8

Let $R(A_1, A_2, \ldots, A_n)$ be a relation and L be a list of elements where each element is either an attribute A_i or an expression $\omega(A_j) \rightarrow B$ with the following restrictions:

- $1 \leq i, j \leq n$
- $\forall i, j,\ i \neq j$
- $\omega \in \{SUM, AVG, MIN, MAX, COUNT\}$
- the SUM, AVG, MIN, MAX, and COUNT operations have their standard SQL interpretation
- $\omega(A_j) \rightarrow B$ denotes its typical meaning, that is, the renaming of $\omega(A_j)$ to B.

The $\gamma_L(R)$ operator is an operation on R with each attribute $A_i \in L$ serving as a grouping attribute (with the set of grouping attributes denoted as GA) and each $\omega(A_j) \rightarrow B$ representing the result of applying the ω operator to A_j relative to GA. The process of constructing this result consists of the following:

- Use the grouping attributes to segment the tuples in R into "groups" where for each $A_i \in GA$, each tuple t_m in a particular group has the same value for A_i. That is, for any two tuples t_m and t_n in a group, $\forall A_i \in GA$, $t_m(A_i) = t_n(A_i)$.

- Then, for each group, do the following:

  - Perform the operations specified by the $\omega(A_j) \rightarrow B$ expressions across all of the tuples in the group and

  - Generate one tuple containing the values for the group's grouping attributes and the results of all of the $\omega(A_j) \rightarrow B$ expressions.

As for an example of an aggregation constraint, we again utilize our Hanyoda Motors scenario to illustrate the applicability of these types of constraints. Consider a simple constraint on the number of tires that a truck has. While uncomplicated, such a constraint is not expressible in the core relational model, and thus we naturally have the need for the γ operation.

To illustrate the usefulness of the γ operation, we modify the Hanyoda Motors Parts entity-relationship diagram in Figure 1.1 to include a *quantity* attribute with the *SubpartS* relationship; Figure 4.3 contains this addition. In Figures 4.4, 4.5, and 4.6, we have extended the tables found in Chapter 1 to include the *quantity* column in the *SubpartS* table; additionally, we have added rows that concern tires. Based on these tables, we can create an expression that reflects the constraint that

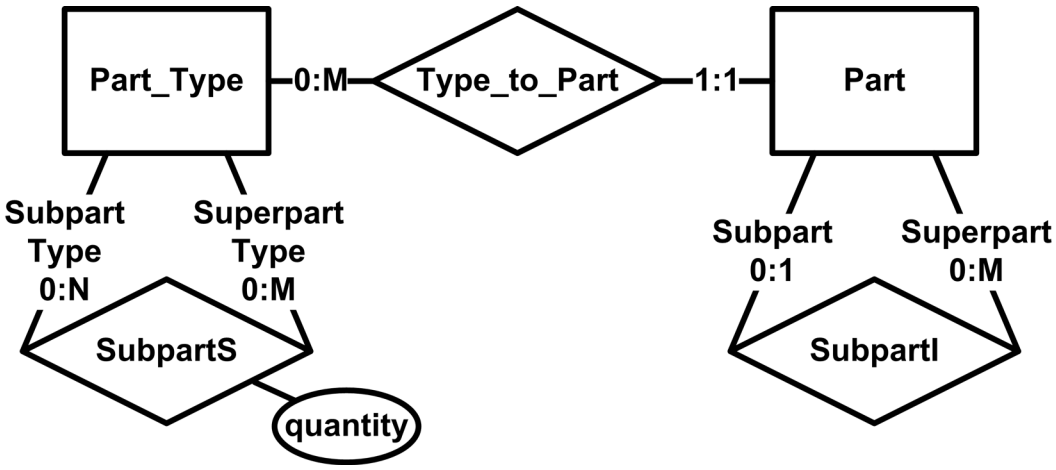

Figure 4.3: Revised Hanyoda Motors Parts ER Diagram

a truck must have exactly 4 tires; in fact, we can express a more general constraint that restricts the part instances in the *Part* table to the quantities specified in the *SubpartS* table. Example 4.3.9 is this expression. It is worth noting that since each part may have several *child* parts (*i.e.*, subparts) but only one *parent* part, the *Part* table contains the part/parent part combination instead of the more commonly seen part/child part combination, and this accounts for the potentially unexpected grouping in Example 4.3.9.

Example 4.3.9 (Part Quantities Constraint)

$\pi_{type\_ID,\, subtype\_ID,\, quantity}(PG) \subseteq SubpartS$ where PG results from the following expression:

$\rho_{PG}(Part \bowtie \gamma_{parent\_part\_ID \rightarrow part\_ID,\, type\_ID \rightarrow subtype\_ID,\, COUNT(part\_ID) \rightarrow quantity}(Part))$

type_ID	description
T1	Full-Size Truck
T2	Compact Truck
E3	Turbo V8 Engine
E5	V8 Engine
E6	V6 Engine
B12	Small Block
B35	Big Block
C45	Extended Cab
C32	Cab
W13	Steel Belted Radial Tire
⋮	⋮

Figure 4.4: Part_Type Table

part_ID	parent_part_ID	type_ID
T4655	NULL	T1
T4656	NULL	T2
E4888	T4655	E5
E1111	T4656	E3
B1234	E4888	B35
B5678	E1111	B35
C9876	T4655	C45
C9877	T4656	C32
W3445	T4655	W13
W3446	T4655	W13
W3447	T4655	W13
W3448	T4655	W13
W3397	T4656	W13
W3398	T4656	W13
W3399	T4656	W13
W3400	T4656	W13
⋮	⋮	⋮

Figure 4.5: Part Table

type_ID	child_type_ID	quantity
T1	E3	1
T1	E5	1
T2	E3	1
T2	E5	1
T2	E6	1
T1	C32	1
T1	C45	1
T2	C32	1
E3	B35	1
E5	B35	1
E6	B12	1
T1	W13	4
T2	W13	4
⋮	⋮	⋮

Figure 4.6: SubpartS Table

Example 4.3.9 illustrates the utility of the γ operation for specifying aggregation-related constraints, and clearly, the opportunity to bring the γ operation to bear on this class of constraints stems from employing views to express constraints. Hence, we have yet another class of constraints that benefits from the duality.

Summary of Relational Extensions

In the preceding, we described three extensions to relational algebra. These three extensions take the form of three operations: δ for metadata (specifically, attribute names), α for transitive closure, and γ for aggregation. Using these operations, we define the following:

Definition 4.3.10 (Extended Relational Algebra (ERA))
The extended relational algebra (ERA) consists of relational algebra extended with the following operations: δ, α and γ.

ERA is QNLOGSPACE in the size of the instance; this follows from results in [CM93]. Furthermore, ERA augmented with set equality/containment (ERAS) captures useful classes of constraints. We have seen these classes throughout this section; as we have demonstrated, ERAS permits a limited form of recursion and allows for the expression of constraints that involve aggregation, the testing of boundary conditions, and schema restrictions. These classes are in addition to the classes that

ERAS subsumes by using relational algebra as its foundation; in other words, ERA can express any constraint expressible in relational algebra.

4.4 Chapter Summary

In this chapter, we present empirical evidence for the first hypothesis in the database systems domain. We supplement Theorem 3.2.1 with empirical results for the well-known constraints and dependencies found in the core relational model. In addition, we discuss relational algebra extensions that permit the expression of richer constraints.

5

Duality in Other Models

In this chapter, we explore models outside the Zachman Framework's What column. We identify two archetypal elements that allow us to argue the existence of the duality in other areas of the framework and thus to continue our hypothesis validation process.

The archetypal framework components that we explore are the process model and the time-based model. The process model resides in the How column, and the time-based model has its origin in the When column. In both cases, we offer representations that allow one to see the manner in which the duality holds.

We then continue our examination by exploring the duality in the cross-column modeling context. The cross-column modeling context concerns modeling that occurs across the columns of the Zachman Framework; in our case, we focus on the interactions between process and time-based models.

5.1 Process Models

We begin with the process model found in the How column and focus largely on the Specify role. Examples of models in other roles in the How column include use case diagrams for the Design role and class diagrams for the Build role[Sof].

Constraints Expressed with Views

We first tackle the capacity of views to express constraints in the process model domain. A process consists of a set of related subprocesses or activities (that is, individual atomic tasks)[Koc05]; the corresponding process model consists of a representation depicting the subprocesses (or activities) as well as the connections between the subprocesses (or activities, respectively). These connections depict control, data, or material flow, and consequently, the literature (such as Pressman in [Pre04] or DeMarco in [DeM79]) also uses "flow" to describe process connections. As flow is directional, a process model is abstractly a directed graph where the direction of a graph arc indicates a dependency of the head process on the tail process. A natural representation of a directed graph is a binary relation, and throughout the remainder of this chapter, we thus characterize process connections as relations and frequently use "relation" in lieu of "process connection." In addition, we do not address the typing of the connections; please review [Koc05] or [MC94] for discussions pertaining

to such typing.

It is also helpful to discuss the nature of a "process view," as we of course are addressing views on process models in this section. Elsewhere in the enterprise architecture literature ([Alb03] for instance), the process view describes at varying levels of abstraction the process-related aspects of an enterprise. In [Zac87, SZ92], the process view is indeed the entire How column. However, we are not discussing process views in this sense; on the contrary, we are investigating views on process models. These views on process models may be some subset of the existing subprocesses and relations in a larger process model; they may also be transformations that create new processes and relations from the existing processes and relations.

Furthermore, we represent views on process models as sets, and the sets themselves fall into two categories: sets of processes (graphically sets of nodes) and sets of relations (graphically sets of edges). In the case of sets of processes, their elements are strictly processes, and the sets carry no notion of relations. As for relations, because relations pertain to processes, the views based on relations also have implicit sets of processes associated with them.

At this juncture, it is useful to return to our Hanyoda Motors scenario from Chapter 1. In Figure 5.1, we depict a high-level order management process that Hanyoda Motors employs. In this process, Hanyoda performs its order entry subprocess upon

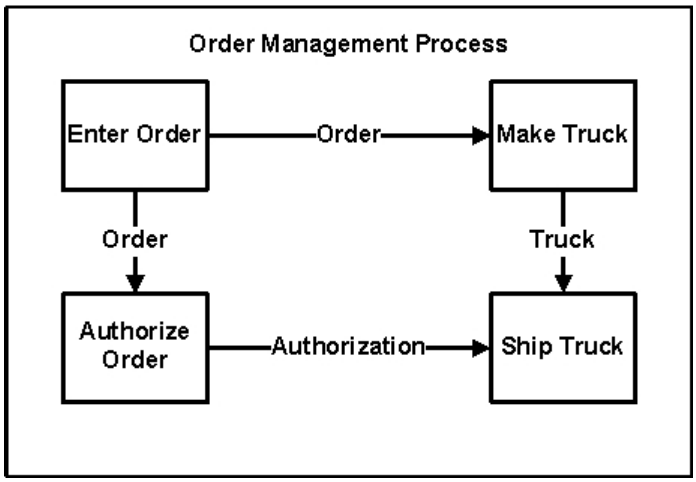

Figure 5.1: Order Management Process

receiving an order, and after order entry, the order is sent to the accounting department for authorization. In addition, production receives the order and manufactures the truck. Once production has made the truck and accounting has authorized the order, the truck can be shipped. Figures 5.2 and 5.3 model the authorization and fulfillment subprocesses, respectively. In the figures throughout this section, we delineate dependencies with directed edges; for example, the directed edge from the "Enter Order" subprocess to the "Authorize Order" subprocess indicates that the "Authorize Order" subprocess depends on the "Enter Order" subprocess. In addition, note that the directed edges (that is, flows) are labeled; these labels indicate the virtual or physical items that flow from a process to one of its related processes.

One method for characterizing the Order Authorization subprocess is to specify

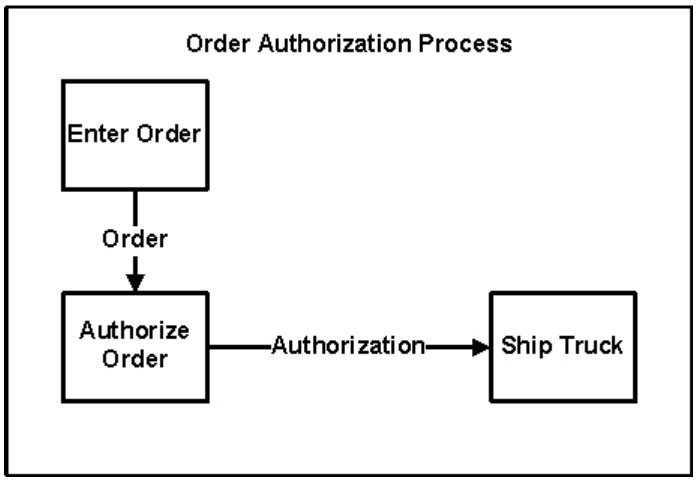

Figure 5.2: Order Authorization Process

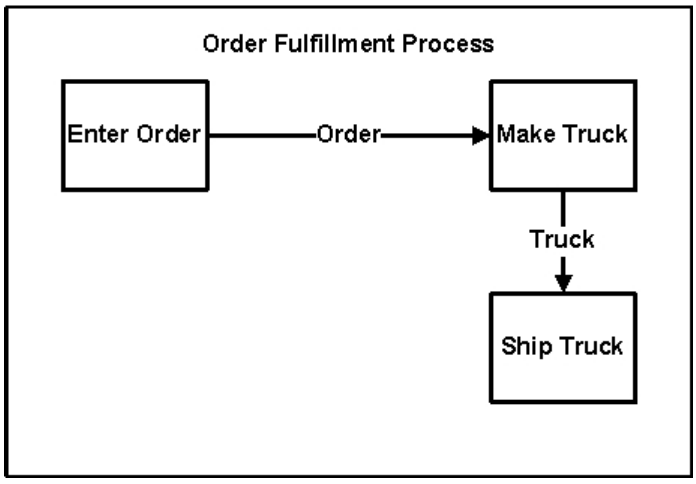

Figure 5.3: Order Fulfillment Process

the relations that constitute it: *Enter_Order_To_Auth_Order* and *Auth_Order_To_Ship_Truck*. The *Enter_Order_To_Auth_Order* relation pairs "Enter Order" processes with their corresponding "Authorize Order" processes; for example, *Enter_Order_To_Auth_Order* relates the "Enter Order 812-XYZ" process to the "Authorize Order 812-XYZ" process. We abbreviate this example (that is, the preceding relation instance) as (ORDER_812-XYZ, AUTH_812-XYZ), and following the same naming convention, we also include (ORDER_317-XYT, AUTH_317-XYT) in *Enter_Order_To_Auth_Order*. Example 5.1.1 formally states the elements in *Enter_Order_To_Auth_Order*.

Example 5.1.1 (Enter Order to Authorize Order Example)

Enter_Order_To_Auth_Order = {(ORDER_812-XYZ, AUTH_812-XYZ),
(ORDER_317-XYT, AUTH_317-XYT)}

As for the *Auth_Order_To_Ship_Truck* relation, it pairs "Authorize Order" processes with their corresponding "Ship Truck" processes; for instance, *Auth_Order_To_Ship_Truck* pairs the "Authorize Order 812-XYZ" process to the "Ship Truck T4655" process. We abbreviate this example as (AUTH_812-XYZ, SHIP_T4655), and use (AUTH_317-XYT, SHIP_T4656) to represent the relation instance that pairs the "Authorize Order 317-XYT" process with the "Ship Truck T4656" process. Example 5.1.2 specifies the members of the *Auth_Order_To_Ship_Truck* relation.

Example 5.1.2 (Authorize Order to Ship Truck Example)

Auth_Order_To_Ship_Truck = {(AUTH_812-XYZ, SHIP_T4655),

(AUTH_317-XYT, SHIP_T4656)}

The Order Fulfillment subprocess also consists of two relations: *Enter_Order_To_Make_Truck* and *Make_Truck_To_Ship_Truck*. Naturally, *Enter_Order_To_Make_Truck* relates the "Enter Order" process with the "Make Truck" process, while the *Make_Truck_To_Ship_Truck* joins the "Make Truck" process with the "Ship Truck" process. Following the conventions established in the preceding paragraphs, we abbreviate the instances in these relations as well; in Examples 5.1.3 and 5.1.4, we use the orders and trucks found in the discussion regarding the Order Authorization subprocess along with the aforementioned conventions to specify the elements in the *Enter_Order_To_Make_Truck* and *Make_Truck_To_Ship_Truck* relations.

Example 5.1.3 (Enter Order to Make Truck Example)

Enter_Order_To_Make_Truck = {(ORDER_812-XYZ, MAKE_T4655),

(ORDER_317-XYT, MAKE_T4656)}

Example 5.1.4 (Make Truck to Ship Truck Example)

Make_Truck_To_Ship_Truck = {(MAKE_T4655, SHIP_T4655),

(MAKE_T4656, SHIP_T4656)}

We now employ the *Enter_Order_To_Auth_Order*, *Auth_Order_To_Ship_Truck*, *Enter_Order_To_Make_Truck*, and *Make_Truck_To_Ship_Truck* relations – in conjunction

with relational composition – to create views defined on the process models in Figures 5.2 and 5.3. Notice that each of these relations is a binary relation, and thus the composition of these relations is a natural operation[Tar41]. We begin by taking the relational composition of *Enter_Order_To_Auth_Order* and *Auth_Order_To_Ship_Truck*. In doing so, we generate a *Order_Auth_Ship* relation that contains instances of corresponding "Enter Order" and "Ship Truck" processes; Example 5.1.5 contains this statement.

Example 5.1.5 (Order Authorization View)

$Order\_Auth\_Ship = Enter\_Order\_To\_Auth\_Order \circ Auth\_Order\_To\_Ship\_Truck =$

{(ORDER_812-XYZ, SHIP_T4655), (ORDER_317-XYT, SHIP_T4656)}

We also take the relational composition of *Enter_Order_To_Make_Truck* and *Make_Truck_To_Ship_Truck*, creating a *Order_Make_Ship* relation that contains instances of corresponding "Enter Order" and "Ship Truck" processes; Example 5.1.6 contains this statement.

Example 5.1.6 (Order Fulfillment View)

$Order\_Make\_Ship = Enter\_Order\_To\_Make\_Truck \circ Make\_Truck\_To\_Ship\_Truck =$

{(ORDER_812-XYZ, SHIP_T4655),(ORDER_317-XYT, SHIP_T4656)}

Examples 5.1.5 and 5.1.6 provide examples of nontrivial views defined on process models, in that they restructure existing relations to create new ones. Implicitly

present in these views are the processes that constitute the new relations; for example, a view containing the relation instance (ORDER_812-XYZ, SHIP_T4655) naturally contains the process instances ORDER_812-XYZ and SHIP_T4655 as well.

Now armed with the requisite views, we define a constraint using these views. As we discussed earlier, only after production has made the truck and accounting has authorized the order can the truck be shipped. Example 5.1.7 expresses this constraint using the views from Examples 5.1.5 and 5.1.6.

Example 5.1.7 (Shipment Authorization Constraint)

Order_Make_Ship $\subseteq$ *Order_Auth_Ship*

We also graphically depict the constraint in Figure 5.4. In that figure, we emphasize the "Enter Order" and "Ship Truck" processes that are present in both views, and in both sides of the figure, we add edges between the aforementioned processes to represent the *Order_Auth_Ship* and *Order_Make_Ship* relations.

Example 5.1.7 illustrates the capacity of views to express constraints in this new context, process models. By continuing to utilize set containment and set equality and capitalizing on the aspect of processes that has its natural representation in relations, we have clear evidence of this facet of the duality in the process model domain.

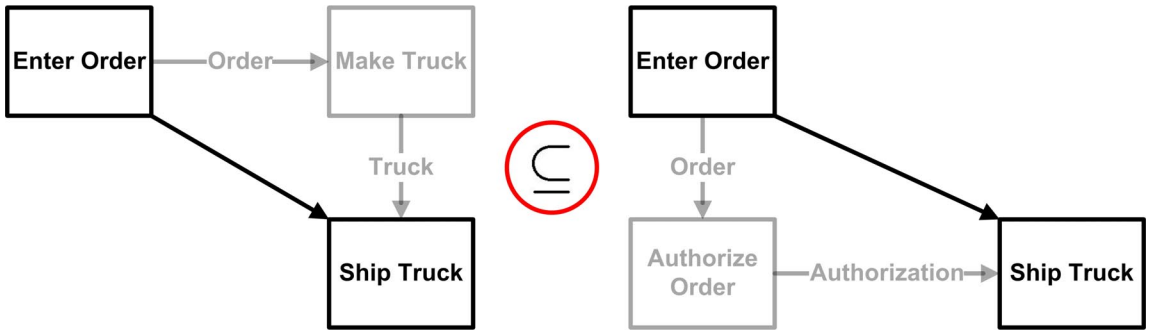

Figure 5.4: Order Process Constraint

Views Expressed with Constraints

As for the other case (views expressed with constraints) of the duality in the process model context, let us begin with a restatement of Example 5.1.7, as expressed in Example 5.1.8.

Example 5.1.8 (Restated Shipment Authorization Constraint)

$Order\_Make\_Ship - Order\_Auth\_Ship = \emptyset$

Reformulating this constraint sets the stage for us to employ a technique similar to the one that we used in the second case of Theorem 3.2.1. Recall that in the second case of Theorem 3.2.1, we essentially removed the quantifiers from the constraints to create views, and in Example 5.1.9, we perform an equivalent action by removing the empty set equality. The resulting expression is a view, pinpointing any order that shipped without the proper authorization.

Example 5.1.9 (Unauthorized Shipped Orders View)

$Order\_Make\_Ship - Order\_Auth\_Ship$

Another constraint to consider is that only ordered trucks are ready for authorization. We express this constraint in Example 5.1.10 using first-order logic.

Example 5.1.10 (Ordered and then Authorized Constraint)

$\forall a, \exists\ o,\ s\ (Auth\_Order\_To\_Ship\_Truck(a,s) \Rightarrow Enter\_Order\_To\_Auth\_Order(o,a))$

Based on Example 5.1.10, we can express a view that identifies all authorized trucks ready for shipment; we do so in Example 5.1.11.

Example 5.1.11 (Authorized Trucks View)

$\{a \mid \exists\ o,\ s\ (Auth\_Order\_To\_Ship\_Truck(a,s) \Rightarrow Enter\_Order\_To\_Auth\_Order(o,a))\}$

Using the same methods that we employed in the core relational model context, we move freely between views and constraints, and this naturally applies beyond these examples to other process model constraints having similar constructions. In the process model context, we thus have compelling evidence of the presence of the facet of the duality that concerns views expressed by constraints.

In this and the preceding subsections, we establish that instances of the duality exist in the cell of the Zachman Framework that is the intersection of the How column and the Specify row, *i.e.*, the conceptual process model context. In the next section,

we transition to another column in the Zachman Framework and investigate the existence of the duality in that context.

5.2 Time-based Models

In this section, we endeavor to discover instances of the duality in the When column of the Zachman Framework. Examples of models in the When column include data flow diagrams in the tradition of [WM91] as well as UML state and sequence diagrams[Sof]; one may denote these as the processing structure and control structure, respectively[fFA].

Constraints Expressed with Views

As we did in the previous section, we begin with constraints expressed with views. The master schedule (or master production schedule) is the prototypical model from the When-Specify cell[1] identified by Zachman in [Zac99]. As discussed in [PS05], a master production schedule is an outline of the products that will be made in each time period (*e.g.*, a week) of a production plan; the APICS Dictionary[CB04] provides a similar definition, specifying that the master production schedule indicates the planned production for individual end items (that is, finished products) in terms

[1] The When-Specify cell is the cell at the intersection of the When column and Specify row.

of dates and quantities. In addition, Portougal[Por06] indicates that the aggregate capacity plan (that is, the planned production capacities for product lines), the inventory levels, and the short-term demand forecast provide the inputs for the master production scheduling process. Table 5.1 shows a master production schedule for Hanyoda Motors. We refer to the period of time covered by a particular master production schedule as its planning period or time horizon; in Hanyoda's master production schedule, this period is 26 weeks.

PRODUCT	WEEK 1	WEEK 2	WEEK 3	...	WEEK 24	WEEK 25	WEEK 26
T1	300	120	190	...	90	80	85
T2	450	260	310	...	240	370	320

Table 5.1: Hanyoda Motors Master Production Schedule

To meet the goals established in the master production schedule, Hanyoda Motors naturally needs more detailed scheduling that addresses daily production. As the authors of [RT06] discuss, several steps follow the creation of the master production schedule in the general scheduling process before actual manufacturing occurs, and the process varies based on the methods used by the parties that are responsible for scheduling.

After the requisite materials and capacity planning[RT06], Hanyoda Motors then arrives at its daily production schedule. This specifies for a particular date and time when each task in the assembly of a particular truck occurs; the schedule also tracks

the part type identifier for the truck involved in each task. In Table 5.2, we list a portion of Hanyoda Motors Daily Production Schedule for August 19, 2006.

DATE_TIME	TASK	TYPE_ID
⋮	⋮	⋮
08/19/2006 8:44	Assemble Chassis	T2
08/19/2006 8:52	Assemble Chassis	T1
⋮	⋮	⋮
08/19/2006 9:15	Add Trim	T2
08/19/2006 9:17	Add Trim	T1
⋮	⋮	⋮
08/19/2006 10:43	Finalize Truck	T2
08/19/2006 10:48	Finalize Truck	T1
⋮	⋮	⋮

Table 5.2: Hanyoda Motors Daily Production Schedule for August 19, 2006

To track its progress toward its goals, Hanyoda Motors naturally wants to correlate its daily production to its master production schedule and constrain its efforts to meet its planned output. An effective approach to defining the desired constraint is to create views on the schedules and then use these views to specify the constraint. As the daily production schedule and the master production schedule have grossly different levels of detail, we need a mechanism to bridge this gap, and to borrow a phrase from the relational database realm, the mechanism we employ is an aggregate view. As its name implies, this type of view utilizes aggregation, and in our case, we limit the aggregation to count (that is, the cardinality of a given set).

We base the first view that we define on the master production schedule; in this

case, we only want our view to pertain to one type of truck and not both types listed in the schedule. For our example, we use the T1 (Full-Size Truck) product, but the same applies to the T2 (Compact Truck) product. Furthermore, we limit the view to only one week in the schedule; for generality, we refer to this as "WEEK n."

We are now prepared to express the view on the master production schedule. In the following, we represent the master production schedule as a two-dimensional array *MPS* with p rows (one row for each product type in the master production schedule) and n columns (one column for each week in the master production schedule). Example 5.2.1 contains *MPS*.

Example 5.2.1 (Master Production Schedule Example)

$$MPS = \begin{pmatrix} 300 & 120 & 190 & \dots & 90 & 80 & 85 \\ 450 & 260 & 310 & \dots & 240 & 370 & 320 \end{pmatrix}$$

To create a view based on *MPS*, we simply provide row and column indices for the array. Since the T1 product corresponds to row 1, we use $p = 1$. As for the column, we use "WEEK 2" as an example and thus specify $n = 2$. Using these indices, we provide Example 5.2.2 to illustrate the resulting view.

Example 5.2.2 (View of Master Production Schedule)

$MPS[1,2] = 120$

Example 5.2.2 returns a value from the master production schedule for the T1 product for the second week in the schedule. We want to now compare that value to

the appropriate information from the daily production schedules. Naturally, we need a week of the daily production schedules; more specifically, we need the schedules for all of the dates that fall in the second week of the master production schedule.

To accomplish this, we choose from the entire collection of daily production schedules the subset of daily production schedules that fall in the week specified in the master production schedule view (in our example, the second week). Furthermore, we specify that the "Finalize Truck" task indicates when a truck has been produced, and we look specifically for the "Finalize Truck" tasks that pertain to the T1 product line. Naturally, an effective method for capturing this information is a view, and as a result, we employ the *Weekly_Production* view. This view ranges over all of the daily production schedules and for a particular product type and week, creates a set containing "Finalize Truck" tasks. In Example 5.2.3, we create the *Weekly_Production* view for the T1 product type and "WEEK 2." The general expression of the *Weekly_Production* view is in Example 5.2.4 and contains a variable T that represents a *TYPE_ID* value.

Example 5.2.3 (Example of Weekly Truck Production Instances View)

Weekly_Production("T1",2) = {"Finalize Truck",...,"Finalize Truck"}

Example 5.2.4 (Weekly Truck Production Instances View)

Weekly_Production(T,w) where $1 \leq w \leq n$ and T = "T1"|"T2"

Consequently, by taking the total number of instances of "Finalize Truck" in the

Weekly_Production view, we have an aggregate view on the daily production schedule that enables us to make the aforementioned comparison. We use the *COUNT* operation to determine that number, and in Example 5.2.5, we add *COUNT* to Example 5.2.4 to create this aggregate view. *COUNT* simply determines the cardinality of a set.

Example 5.2.5 (Weekly Truck Production Totals View)

$COUNT(Weekly\_Production(\mathrm{T},w))$

We now have the two views that we need to create the desired constraint. Example 5.2.7 contains the expression that specifies that the daily production schedules for a particular product and week must meet the master production schedule's planned production for that same product and week. We first define the *TYPE_ID_ARRAY* array that contains the *TYPE_ID* values in the order in which they appear in the master production schedule; Definition 5.2.6 contains this definition.

Definition 5.2.6 (*TYPE_ID_ARRAY* Definition)

$TYPE\_ID\_ARRAY = (\text{“T1”}\ \text{“T2”})$

Example 5.2.7 (Daily Truck Production Schedule Constraint)

$MPS[i,j] = COUNT(Weekly\_Production(\mathrm{T},j))$ where $TYPE\_ID\_ARRAY[i] = \mathrm{T}$, $1 \leq i \leq p$, and $1 \leq j \leq n$

We extend Example 5.2.7 to include all product types and weeks in the master

production schedule. Example 5.2.8 specifies this extension.

Example 5.2.8 (Daily Total Production Schedule Constraint)

$\forall 1 \leq i \leq p,\ 1 \leq j \leq n,\ MPS[i,j] = COUNT(Weekly\_Production(\mathrm{T},j))$ where $TYPE\_ID\_ARRAY[i] = \mathrm{T}$

Examples 5.2.7 and 5.2.8 demonstrate the capacity of views to express constraints in yet another context, time-based models. Previously, we have used set containment and set equality, but in this context, we employ a different technique – an aggregate view – to calculate a number that we then use in a comparison with another number.

Views Expressed with Constraints

Regarding the other case (views expressed with constraints) of the duality in the time-based model context, we build on Example 5.2.7, and from it, we create a series of views. In this endeavor, we use the variables T and j (as in "WEEK j") that appear in Example 5.2.7 to specify the contents of the first view. The view contains at most one pair T and j, and it only contains this pair if the constraint is met. Example 5.2.9 holds this view expression.

Example 5.2.9 (View Based on Daily Truck Production Schedule Constraint)

$\{(\mathrm{T},j) \mid MPS[i,j] = COUNT(Weekly\_Production(\mathrm{T},j))$ where $TYPE\_ID\_ARRAY[i] = \mathrm{T}$ and $1 \leq j \leq n\}$

In a sense, Example 5.2.9 corresponds to the second case of Theorem 3.2.1, in that the variables in Example 5.2.7 are "bound" variables that become "free" in Example 5.2.9. Alone, the view in Example 5.2.9 is sufficient evidence that views can be expressed with the aid of constraints in the time-based context. However, we can use Example 5.2.9 in another expression that serves to indicate instances where the constraint fails; Example 5.2.10 contains this expression.

Example 5.2.10 (Violator Identifying View Based on Daily Truck Production Schedule Constraint)

$\{(\mathrm{T},j) \mid MPS[i,j] \neq COUNT(Weekly\_Production(\mathrm{T},j))$ where $TYPE\_ID\_ARRAY[i] = \mathrm{T}$ and $1 \leq j \leq n\}$

If the constraint fails, Example 5.2.10 generates the violators; if the constraint holds, it generates an empty set. The capacity to include Example 5.2.9 in another view (Example 5.2.10) provides additional empirical evidence of the presence of the duality in the time-based context. Example 5.2.10 also showcases yet another example of the effectiveness of the duality in pinpointing constraint violators.

5.3 Cross-column Models

In this section, we examine the duality in the cross-column modeling context, and in this context, we focus on the interactions between process and time-based models. As we did in previous sections, we begin with the use of views in expressing constraints; in this particular section, we employ process and time-based model views in time-based constraints. We then demonstrate the use of time-based constraints in the construction of views on process and time-based models.

Constraints Expressed with Views

In this subsection, we investigate instances of the duality by utilizing process and time-based model views in time-based constraints. We continue to employ our Hanyoda Motors example. In Figure 5.5, we depict the truck assembly process and its five major subprocesses. In that figure, we delineate dependencies (that is, relations) with directed edges, as is the accepted convention and the standard to which we conform in this chapter; for example, the directed edge from the "Assemble Body" process to the "Paint Body" process indicates that the "Paint Body" process depends on the "Assemble Body" process. In addition, notice that the "Finalize Truck" process depends on the "Add Trim" and "Assemble Chassis" processes.

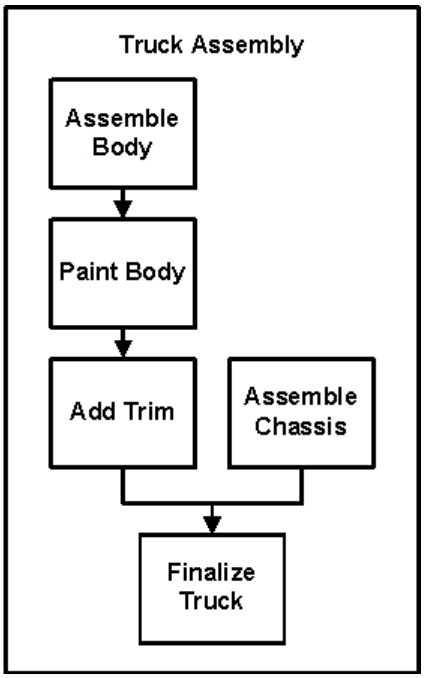

Figure 5.5: Process Flowchart

In addition to the requisite materials and capacity planning[RT06], Hanyoda Motors generates a model that describes the temporal sequence that must be obeyed during truck assembly. Known as an activity or task network diagram[Pre04], it can range in detail from a high-level depiction such as the process model found in Figure 5.5 to a diagram specifying the ordering of the atomic tasks in a particular process. For our purposes, a high-level depiction is suitable, and thus we employ Figure 5.5 as Hanyoda Motors' task network diagram. Henceforth, we refer to the processes in Figure 5.5 as tasks.

The task network diagram impacts the daily production schedule in Table 5.2. In Table 5.3, we revise the Hanyoda Motors Daily Production Schedule for August 19, 2006 (that is, Table 5.2) to include specific part identification numbers for the trucks; the tasks listed in Table 5.3 correspond to the tasks in Figure 5.5.

DATE_TIME	TASK	TYPE_ID	PART_ID
⋮	⋮	⋮	⋮
08/19/2006 8:44	Assemble Chassis	T2	T4656
08/19/2006 8:52	Assemble Chassis	T1	T4656
⋮	⋮	⋮	⋮
08/19/2006 9:15	Add Trim	T2	T4656
08/19/2006 9:17	Add Trim	T1	T4656
⋮	⋮	⋮	⋮
08/19/2006 10:43	Finalize Truck	T2	T4656
08/19/2006 10:48	Finalize Truck	T1	T4656
⋮	⋮	⋮	⋮

Table 5.3: Revised Hanyoda Motors Daily Production Schedule for August 19, 2006

A simple constraint on the daily production schedule is that for each product (that is, a truck), no task that follows another task in the task network diagram can precede that same task in the daily production schedule. For example, since the "Finalize Truck" task follows the "Add Trim" task in the task network diagram, it cannot precede the "Add Trim" task in the daily schedule. Obviously, this constraint does not have to hold for different products (*e.g.*, trucks T4655 and T4656); in fact, given the nature of activities on an assembly line, it is very likely optimal that the constraint does not hold. However, for an individual product, the constraint is mandatory.

The first step in using views to express this constraint involves the task network diagram. In this section, of primary importance is the order of the abstract tasks in the task network diagram (*i.e.*, the tasks specified in the model); the use of abstract tasks in this case is in sharp contrast to earlier in this chapter when our focus was on process instances. Toward that end, we create a *Depends_On* relation that captures the order of the abstract tasks. Table 5.4 provides the *Depends_On* relation for the tasks found in Figure 5.5. Notice that the tuples in the *Depends_On* relation in Table 5.4 contain the abstract tasks, not instances. Furthermore, we use the PRECEDING_TASK and SUCCEEDING_TASK attributes in the *Depends_On* relation to distinguish between the tasks involved in a particular dependency.

We next take the transitive closure of the *Depends_On*. Following the convention

employed throughout this document, we represent transitive closure with the α operation; consequently, we express the transitive closure of *Depends_On* as α(*Depends_On*). Taking the transitive closure of this relation is necessary to create explicit dependencies between all of the tasks in the task network diagram.

We also utilize the daily production schedule from Table 5.3, representing it as the *DPS* relation with attributes DATE_TIME, TASK, TYPE_ID, and PART_ID. We then employ domain relational calculus notation to express a view that utilizes the *DPS* relation. This expression appears in Example 5.3.1, and we refer to the expression as *BEFORE*.

Example 5.3.1 (Task Succession View)

$BEFORE = \{(a_p, a_s) \mid (d_p,a_p,i_p,p_p) \in DPS \wedge (d_s,a_s,i_s,p_s) \in DPS \wedge p_p = p_s \wedge d_p < d_s\}$

We are now prepared to express the constraint. It appears in Example 5.3.2 and uses the expression in Example 5.3.1 as well as α(*Depends_On*).

Example 5.3.2 (Task Succession Constraint)

$BEFORE \subseteq \alpha(Depends\_On)$

Simply stated, the constraint specified in Example 5.3.2 aims to eliminate the case where a task in the schedule disobeys the task network diagram. Moreover, Example 5.3.2 illustrates the capacity of views to express constraints in yet another

context, cross-column models. We next address the other aspect of the duality – the ability to express views with constraints – in this same context.

Views Expressed with Constraints

In this subsection, we endeavor to discover instances of the duality by utilizing time-based constraints in the construction of views on process and time-based models; an instance of this quickly follows from the example in the previous subsection. Let us begin with a restatement of Example 5.3.2, as expressed in Example 5.3.3.

Example 5.3.3 (Task Succession Constraint Using Empty Set)

$BEFORE - \alpha(Depends\_On) = \emptyset$

We next repeat our actions from earlier in this chapter and remove the empty set equality. The resulting expression is a view that pinpointing the pairs of tasks that do not obey the task network diagram. We state this view in Example 5.3.4.

Example 5.3.4 (View Based on Task Succession Constraint)

$BEFORE - \alpha(Depends\_On)$

Example 5.3.4 serves as a clear illustration of a cross-column modeling situation where one uses a constraint in the construction of a view. However, it has limited utility as a mechanism to identify the constraint violators; this limitation stems from

its inability to determine to which truck a pair of tasks belongs. In turn, this inability results from the abstract nature of the *Depends_On* relation. That is, the *Depends_On* relation has as its domain the abstract tasks of Figure 5.5, and an abstract task does not pertain to a specific truck. Consequently, the *Depends_On* relation has no notion of pairs of tasks for a *specific* truck (*e.g.*, truck T4655); it simply contains the abstract dependencies.

Because of the nature of the *Depends_On* relation and the manner in which we specify the constraint (Example 5.3.2), the *BEFORE* view only contains pairs of tasks, and consequently, the information that matches tasks with trucks is lost. This motivates us to find another method for expressing this constraint, and we offer an alternative in Example 5.3.6. This example utilizes the *EARLY* view from Example 5.3.5.

Example 5.3.5 (Violator Identifying View Based on Task Succession Instances)

$EARLY = \{S \mid S \in DPS \wedge \exists P \in DPS, \exists D \in Depends\_On(P.\text{PART\_ID} = S.\text{PART\_ID} \wedge P.\text{TASK} = D.\text{PRECEDING\_TASK} \wedge S.\text{TASK} = D.\text{SUCCEEDING\_TASK} \wedge P.\text{DATE\_TIME} > S.\text{DATE\_TIME})\}$

Example 5.3.6 (Task Succession Instances Constraint)

$EARLY = \emptyset$

In the *EARLY* view, we utilize tuple relational calculus to identify the set of tuples

from *DPS* (*i.e.*, tasks in the daily production schedule) that disobey the constraint; adding the empty set equality in Example 5.3.6 seeks to enforce the constraint. Moreover, we overcome the limitations of Example 5.3.2 by using the tuple variable S that contains date/time, task, type identifier, and part identifier information; consequently, we are able to identify the actual truck-task pairs that violate the constraint.

5.4 Chapter Summary

In this chapter, we explore models outside the Zachman Framework's What column and identify two archetypal elements (process and time-based models) that allow us to argue the existence of the duality in other areas of the framework. We then continue our investigation of the duality in the cross-column modeling context. We focus on the interactions between process and time-based models and provide evidence of the existence of the view↔constraint duality in this context as well.

PRECEDING_TASK	SUCCEEDING_TASK
Assemble Body	Paint Body
Paint Body	Add Trim
Add Trim	Finalize Truck
Assemble Chassis	Finalize Truck

Table 5.4: The *Depends_On* relation

6

Duality Use in Practical Contexts

In this chapter, we delve into the second hypothesis: the view↔constraint duality is a conceptually tractable and useful tool in the development of information systems. As a tool, the duality is conceptually tractable in that it utilizes a familiar medium, views. As for its usefulness, this stems from the capacity of the duality to provide solutions to practical problems. We return to the relational database context, where formal mechanisms are available. The conventions that we employ throughout this chapter follow.

Conventions 6.0.1 (Chapter 6 Conventions)

We assume the following:

> $V(R_1, \ldots, R_k) \subseteq W(S_1, \ldots, S_l)$ is a constraint expression where V and W are views, $\{R_1, \ldots, R_k\}$ are the relation occurrences in V, and $\{S_1, \ldots, S_l\}$ are the relation occurrences in W. This is a syntactic convention, in that we use *relation*

occurrence to indicate that a relation may occur more than once in $R_1, \ldots, R_k$ or $S_1, \ldots, S_l$. The order of the R_i or S_j is that found in a left to right scan of the expressions for V or W, respectively. In Section 6.1, we assume without loss of generality that no attribute name is of a form that conflicts with renaming done in that section.

V' and W' are views whose definitions will be provided when necessary.

Given instances $\mathbf{r}_1, \ldots, \mathbf{r}_k$, and $\mathbf{s}_1, \ldots, \mathbf{s}_l$ of $R_1, \ldots, R_k$, $S_1, \ldots$, and S_l, respectively; $\mathbf{r}_V$, $\mathbf{s}_W$, $\mathbf{r}_{V'}$, and $\mathbf{s}_{W'}$ are instances resulting from the evaluation of V, W, V', and W', respectively. Note that, if R_i and R_j have the same relation name, then $\mathbf{r}_i = \mathbf{r}_j$ and similarly for the $\mathbf{s}_i$; if R_i and S_j have the same name, then $\mathbf{r}_i = \mathbf{s}_j$.

$V(R_1, \ldots, R_k) \subseteq W(S_1, \ldots, S_l)$ may be expressed as $V \subseteq W$ if the relation occurrences are understood. Furthermore, $V \subseteq W$ denotes that $\mathbf{r}_V \subseteq \mathbf{s}_W$, and this extends to all expressions involving views and the $\subseteq$ operation.

We extend π to operate on individual tuples. That is, $\{\pi_X(t)\} = \pi_X(\{t\})$.

6.1 Recovering Constraint Violators

In this section, we investigate the general case where the boolean nature of a view-based constraint expression results in the loss of information that otherwise would be useful in rectifying violations of the constraint. When a constraint $M \subseteq N$ fails, we assume that the failure is due to violators in M, that is, elements of $M - N$.

We start by restating Example 4.3.9 – the constraint that restricts the part instances in the *Part* table to the quantities specified in the *SubpartS* table – in Example 6.1.1.

Example 6.1.1 (Part Quantities Constraint)

$\pi_{type\_ID,\, subtype\_ID,\, quantity}(PG) \subseteq SubpartS$ where PG results from the following expression:

$\rho_{PG}(Part \bowtie \gamma_{parent\_part\_ID \rightarrow part\_ID,\, type\_ID \rightarrow subtype\_ID,\, COUNT(part\_ID) \rightarrow quantity}(Part))$

Example 6.1.1 does not permit the identification of potential constraint violators, as that level of detail is lost when projecting into a schema that matches the schema of the *SubpartS* relation. However, we can directly transform the expression into one that allows the quick identification of the constraint violators; we provide this transformed expression in Example 6.1.2.

Example 6.1.2 (Part Quantities Constraint with Violators Identified)

The rewritten constraint is:

$$\pi_{part\_ID,\, type\_ID,\, subtype\_ID,\, quantity}(PG) \subseteq (\pi_{part\_ID,\, type\_ID,\, subtype\_ID}(PG) \bowtie SubpartS)$$

The violators are:

$$\pi_{part\_ID,\, type\_ID,\, subtype\_ID,\, quantity}(PG) - (\pi_{part\_ID,\, type\_ID,\, subtype\_ID}(PG) \bowtie SubpartS)$$

A natural question arises from Example 6.1.2 that concerns generality. Specifically, is it possible to express a general rule for rewriting view-based constraints that allows the constraints in their modified forms (through the techniques employed throughout this document such as removing the empty set equality) to identify explicitly the tuples that violate the constraints? As we see in Example 6.1.1 and in other instances throughout this document, using views to express constraints can create situations where information[1] is lost that otherwise would be important in leveraging the views to identify the violators, and we seek to eliminate this information loss. As this loss is a product of the manner in which one specifies the constraint, we use Constraint Specification Loss (CSL) to denote instances when this information loss occurs.

In the pursuit of CSL elimination, it is crucial to the inquiry that the views participating in a constraint expression have *schematic conformity*. We formally define *schematic conformity* in Definition 6.1.3; this definition assumes the named perspective on the attributes of a relation as discussed by Abiteboul, Hull, and Vianu in [AHV95] as opposed to one based on attribute types. *Schematic conformity* permits

[1] We do not use *information* in its formal, entropy-based meaning. An investigation of the application of information-theoretic measures such as entropy to this problem is warranted. However, such an inquiry is not germane to the current discussion, and thus we do not pursue it at this time.

the set comparison operations that are at the core of our framework and follows the standard named relational algebra conventions for union, intersection, and set difference.

Definition 6.1.3 (Schematic Conformity)

V and W have *schematic conformity* iff $sort(V) = sort(W)$.

To assist in recovering constraint violators, we introduce Theorem 6.1.4.

Theorem 6.1.4 (RA View-based Constraint Rewriting)

Let $V(R_1, \ldots, R_k) \subseteq W(S_1, \ldots, S_l)$ be a constraint expression where V and W have *schematic conformity* and are relational algebra (RA) expressions. Then, there exists V' and W' such that the following are true for all instances of R_1, ..., R_k, S_1, ..., and S_l:

- V' and W' have *schematic conformity*,
- the attributes of V' cover the primary keys of $R_1, \ldots, R_k$ (possibly renamed) with the exception of any primary key that appears only on the right-hand side of a "$-$" operation,
- $V = \pi_{sort(V)}(V')$, and
- $V \subseteq W$ iff $V' \subseteq W'$.

Proof:

Construct V' as follows:

(a) With the exception of any subexpression of V that appears on the right-hand side of a "$-$" operation, replace each projection in V with a renaming operation that renames the attributes that are dropped in the projection. Specifically, replace the expression $\pi_X(E)$ where E is a subexpression and X is the set of attributes in the original projection with the expression $\rho_{Y_1 \rightarrow \mathtt{R}\eta(i_1)\_Y_1,\ldots,Y_n \rightarrow \mathtt{R}\eta(i_n)\_Y_n}(E)$ where $sort(E) - X = \{Y_1, \ldots, Y_n\}$, η(x) indicates the decimal representation of x, and i_m is the least value v such that

- R_v is in the scope of E,
- R_v contains attribute Y_m and
- that occurrence of Y_m has not been renamed or shielded by a "$-$" operation in a subexpression within E.

(b) Resolve all schematic conformity discrepancies between set operations created by the removal of projections; handle these discrepancies recursively in the following manner. In each case, consider a subexpression $E' \triangle F'$ that is constructed from the subexpression $E \triangle F$ of V where $\triangle \in \{$"$\bigcup$", "$\bigcap$", "$-$"$\}$ and E' and F' are respectively E and F rewritten recursively.

- When $\triangle =$ "$\bigcup$" and for each attribute B in sort(E') - sort(F') or in sort(F') - sort(E'), use Cartesian product to add the attribute B to

F' or E', respectively such that B has a NULL value in all tuples (also known as an `OUTER UNION` operation[EN06]). Note that any such B is of the form $\mathtt{R}\eta(i_j)\_Y_k$ in accordance with the earlier renaming.

- When $\triangle =$ "$\bigcap$", replace $\bigcap$ by a natural join. Since renamed attributes are unique to either E' or F', only attributes appearing in E (and F by symmetry) occur in equality conditions implied by the natural join.
- When $\triangle =$ "$-$", F' is simply F. There are no attributes in F' that do not also occur in E and hence in E'. Attributes in E' but not in F are handled by defining F' as $E' \bowtie F$.

Next, construct W' by joining V' to W using a natural join: $W' = V' \bowtie W$.

As a result of $V' \bowtie W$, $sort(V') = sort(W')$. Thus W' conforms to V'. As for the attributes of V' covering the primary keys of $R_1, \ldots, R_k$ with the exception of any primary key that appears on the right-hand side of a "$-$" operation, the construction of V' dictates that $sort(V')$ consists of all attributes in $sort(R_1)$ $\cup$ ... $\cup$ $sort(R_k)$ (with appropriate renaming as described in the preceding) excluding attributes that appear on the right-hand side of a "$-$" operation. Thus $sort(V')$ must contain the primary key attributes of $R_1, \ldots, R_k$ with the exception of primary key attributes involved in the right-hand side of a set difference operation.

It thus remains to prove $V = \pi_{sort(V)}(V')$ and $V \subseteq W$ iff $V' \subseteq W'$. Henceforth,

without loss of generality, we use $\mathbf{r}_V$, $\mathbf{s}_W$, $\mathbf{r}_{V'}$, and $\mathbf{s}_{W'}$ as instances representative of the evaluation of V, W, V', and W', respectively across all instances of R_1, ..., R_k, S_1, ..., and S_l. We begin by stating that if $\mathbf{r}_V = \emptyset$, then $V \subseteq W$ iff $V' \subseteq W'$ is vacuously true. Otherwise, we proceed by specifying Lemmas 6.1.5 and 6.1.6.

Lemma 6.1.5

V is algebraically equivalent to $\pi_{sort(V)}(V')$ and hence $\mathbf{r}_V = \pi_{sort(V)}(\mathbf{r}_{V'})$

Proof:

Because only rewritten variables are dropped in the projection $\pi_{sort(V)}$, that projection may be distributed over operations in V' until the appropriate renamings are found and the renamings removed. Thus $\pi_{sort(V)}(V')$ is algebraically equivalent to V.[2] □

Lemma 6.1.6

$\pi_V(\mathbf{s}_{W'}) = \pi_V(\mathbf{r}_{V'}) \cap \mathbf{s}_W$, hence $\pi_V(\mathbf{s}_{W'}) \subseteq \pi_V(\mathbf{r}_{V'})$ and $\pi_V(\mathbf{s}_{W'}) \subseteq \mathbf{s}_W$.

Proof:

By the construction of W' ($W' = V' \bowtie W$), the lemma must hold. □

[2]The algebraic rule is $\pi_X(\rho_{A \to B}(\mathbf{r})) = \pi_X(\mathbf{r})$ when $B \notin X$.

Lemma 6.1.5 proves that $V = \pi_{sort(V)}(V')$. We now must demonstrate that the set containment/equality relationships may be inferred in both directions.

Case 1: $\mathbf{r}_V \subseteq \mathbf{s}_W \Rightarrow \mathbf{r}_{V'} \subseteq \mathbf{s}_{W'}$

By Lemma 6.1.5, any $t' \in \mathbf{r}_{V'}$ corresponds to $t \in \mathbf{r}_V$ such that $\{t\} = \pi_{sort(V)}(\{t'\})$ and by the case hypothesis, $t \in \mathbf{s}_W$. Thus t joins with t' in the expression $V' \bowtie W$ and hence $t' \in \mathbf{s}_{W'}$.

Case 2: $\mathbf{r}_V \subseteq \mathbf{s}_W \Leftarrow \mathbf{r}_{V'} \subseteq \mathbf{s}_{W'}$

By Lemma 6.1.5, $\mathbf{r}_V = \pi_{sort(V)}(\mathbf{r}_{V'})$. By Lemma 6.1.6 and the case hypothesis, $\mathbf{r}_{V'} = \mathbf{s}_{W'}$, and hence $\pi_{sort(V)}(\mathbf{r}_{V'}) = \pi_{sort(V)}(\mathbf{s}_{W'})$. Again by Lemma 6.1.6, $\pi_{sort(V)}(\mathbf{s}_{W'}) \subseteq \mathbf{s}_W$, and the case is proved. □

By virtue of Theorem 6.1.4, we have the capacity to recover constraint violators in view-based constraint expressions when the views themselves are relational algebra expressions. Thus, when encountering CSL involving views consisting of relational algebra expressions, we have an effective mechanism for eliminating the loss of information.

When the constraint is $V = W$, violators may appear on either side of the expression. Thus, we must separate $V = W$ into two cases, $V \subseteq W$ and $V \supseteq W$. We then apply Theorem 6.1.4 to each case to get $V' \subseteq W'$ and $V'' \supseteq W''$, respectively. The conjunction of these is equivalent to $V = W$, but it is more effective to look for

violators independently.

An additional observation regarding Theorem 6.1.4 is helpful. Our construction for $E' - F'$ is very inefficient, in that it inflates a set merely to create the difference. A form equivalent to $E' - (E' \bowtie F')$ that does not expand F' is $E' \bowtie (E - F)$. Moreover, we may simplify this to $E' \bowtie (\pi_{sort(E)}(E') - F)$, thus avoiding the need to compute both E and E'.[3]

It is also helpful to consider views whose expressions use aggregation and transitive closure in addition to the core relational algebra. As a matter of tractability, we must treat the relations resulting from aggregate and transitive closure operations as atomic; that is, for rewriting purposes, we do not descend into the subexpressions *dominated* (as described by Zanibbi, Blostein, and Cordy in [ZBC02]) by aggregation and transitive closure operations. We designate the relations that result from the aggregate and transitive closure operations as intermediate relations, and henceforth, we focus on the schemas for the intermediate relations.

As a consequence of treating the intermediate relations as atomic, it is paramount that their schemas already contain the relevant primary key attributes for the relations that appear in the expressions that serve as input to the aggregation and transitive closure operations. Otherwise, recovering the identity of a constraint violator is not

[3] $E' \bowtie (\pi_{sort(E)}(E') - F)$ is an instance of the `OUTER DIFFERENCE` operation, signified as $\pm$ and defined as $P \pm Q = P \bowtie (\pi_{sort(Q)}(P) - \pi_{sort(P)}(Q))$ where P and Q are relations. Since E and F have the same schema in our example, only the $\pi_{sort(E)}$ projection is necessary.

possible when relevant primary key attributes missing from the intermediate relations' schemas.

On this topic, consider a constraint that all active classes must have at least 10 students enrolled. In this case, the constraint has violations associated with classes, not individual students. It is likely that the underlying relational database schema implements the relationship between students and classes by utilizing a table whose primary key is a composite consisting of a student identifier and a class identifier. Assuming we indeed implement the table in this manner, the constraint violators would be class identifiers, not the entire primary key.

As an additional example, consider a constraint that all active classes must have enrollments that do not exceed the seating limits of their assigned classrooms. This constraint has violations that are associate with classes if the constraint is checked in *batch mode* (that is, the constraint is checked against the entire database instance). However, if the constraint is checked in *transaction mode* (that is, the constraint is checked when a new tuple is inserted), the student who puts the count over the limit would be the violator.

In general, one would expect that the relevant keys would be among the grouping (*i.e.*, GROUP BY) attributes, at least in *batch mode*. Thus, we add to this discussion a notion of *relevant* primary keys. For rewriting according to Theorem 6.1.4 to occur, we naturally must determine the relevant primary keys. As relevance is a semantic

consideration and depends in some sense on the meaning of the expressed constraint, it becomes difficult to stipulate an automated procedure, and thus we see the need for user intervention in this determination process. In a *transaction mode* context, *strength reduction* provides another approach, as we discuss in the next section.

We can also apply this approach in cases where a view contains both aggregation and transitive closure; in these cases, we treat the least nested aggregation or transitive closure operation as creating an intermediate relation and then determine if the intermediate relation contains the relevant primary keys. We then can utilize the results from Theorem 6.1.4 to enable the recovery of violators in view-based constraints that utilize the extended relational algebra under the assumption that the views have the relevant keys in their schemas.

This same technique applies when constraints involve self-referential relationships, as implemented by $parent\_part\_ID$ in the $Part$ relation of Example 6.1.1. A self-referential relation has a primary key attribute ($part\_ID$ in the example) and a foreign key attribute (*e.g.*, $parent\_part\_ID$) that refers to the primary key. If the foreign key is a grouping attribute of the aggregation, then it may be possible to recover the constraint violators. In effect, we treat the self-referential foreign key as a primary key and thus proceed to use the means established in Theorem 6.1.4. Nevertheless, this is another circumstance truly needing user intervention to ensure correctness.

The naturalness of this process is seen in Examples 6.1.1 and 6.1.2, where the

derived relation PG encapsulates the aggregate operation (and an additional join to provide the proper $type\_ID$). The process actually results in the expression stated in Example 6.1.7.

Example 6.1.7 (Part Quantities Constraint Resulting from Theorem 6.1.4)

$PG \subseteq (PG - SubpartS)$

In place of the expression from Example 6.1.7 that the aforementioned procedure dictates, we utilize the expression from Example 6.1.2 that refrains from joining the *quantity* columns from PG and $SubpartS$. This is a slight optimization, and certainly other optimizations may be applicable. In fact, we can leverage the well-known heuristics and optimizations of the relational model by utilizing view-based constraints. In addition, it is worthwhile to state explicitly that the set of violators consists of $PG \subseteq (PG - SubpartS)$.

6.2 Triggers

In this section, we discuss a use of the duality involving triggers. As we saw in Chapter 2, the mechanism for implementing triggers is typically powerful, permitting use of general purpose programming languages that are Turing-complete. Considering the powerful expressiveness of triggers, it is difficult to posit views as a

mechanism that can effectively express all triggers. However, triggers that are computationally tractable can be often derived from view-based constraints using *strength reduction*[ASU86], a notion that has long been used in view maintenance[KP81]. Several authors, including [CW90, CFPT94, TG01], have explored the automated generation of triggers from constraints; Ceri *et al.*[CFPT94] even discuss the use of views in their constraint specifications. Using strength reduction in the context of the duality, though, is novel.

To prepare for our discussion of view-based constraints in the context of triggers, we must appeal to the database management system in which we implement the trigger. In commercial database management systems, there are often methods for the trigger to inspect the tuple before its insertion; Oracle provides the `OLD` and `NEW` pseudorecords for these occasions[UM04]. The `OLD` pseudorecord represents the existing tuple that we are deleting or updating (in the case of `DELETE` or `UPDATE`, respectively), while the `NEW` pseudorecord represents the new tuple that results from an `INSERT` or `UPDATE`.

We begin by examining the components of triggers (that is, events, conditions, and actions) individually and discussing the manner in which views may address them. We first discuss events; unfortunately, views certainly do not possess a faculty for representing events' temporal aspects. Furthermore, events concern data manipulation operations (such as insert, delete, and update) while views are declarative.

Consequently, directly specifying the event component of triggers falls outside the scope of views. We can, however, capture the result of these events by virtue of the limited number of event types: insert, update, and delete. Consider $\mathbf{r}$ and $\mathbf{r}'$ as the pre-event and post-event instances of a relation R on which a trigger resides, with tuples $\texttt{NEW} \in (\mathbf{r}' - \mathbf{r})$ and $\texttt{OLD} \in (\mathbf{r} - \mathbf{r}')$. In this context, we characterize each of the event types as follows:

$\texttt{INSERT}$: $\mathbf{r}' = \mathbf{r} \cup \{\texttt{NEW}\}$

$\texttt{DELETE}$: $\mathbf{r}' = \mathbf{r} - \{\texttt{OLD}\}$

$\texttt{UPDATE}$: $\mathbf{r}' = \mathbf{r} - \{\texttt{OLD}\} \cup \{\texttt{NEW}\}$

In the sequel, we assume a constraint holds prior to the occurrence of an event. We characterize this as a precondition on the trigger.

As for conditions, views can naturally express boolean-valued queries, and consequently, views can account for the conditions present in triggers. In the case of actions, their procedural nature creates some difficulty for using views to express them. However, when the result of a action is a single relation, a view can obviously express this type of result. The result of an action may also impact multiple relations, and this case naturally calls for multiple views. Hence, views can express the result(s) obtained from a trigger's action, but such an approach may require employing multiple views, depending on the composition of the action. Furthermore, while

a view (or views) can capture the final result of an action (or actions), the action itself may utilize non-relational structures or control statements (such as for loops) in the intermediate steps of its procedure, and such entities are naturally beyond the purview of views.

In some cases, one can derive conditions and actions from the view-based constraints using the aforementioned strength reduction. For example, consider the foreign key constraint that A_i in $\boldsymbol{r}$ is a foreign key that refers to B_j in $\boldsymbol{s}$. As noted in Chapter 4, the view form is $\pi_{A_i}(\mathbf{r}) \subseteq \pi_{B_j}(\mathbf{s})$, or equivalently, $\pi_{A_i}(\mathbf{r}) - \pi_{B_j}(\mathbf{s}) = \emptyset$.

Now consider the effect of inserting a single tuple `NEW` into $\boldsymbol{r}$ (the `TRIGGER`'s event). The constraint may be rewritten to compensate for this new tuple, resulting in the following:

$$\begin{aligned} \pi_{A_i}(\mathbf{r} \cup \{\texttt{NEW}\}) - \pi_{B_j}(\mathbf{s}) &= (\pi_{A_i}(\mathbf{r}) \cup \pi_{A_i}(\{\texttt{NEW}\})) - \pi_{B_j}(\mathbf{s}) \\ &= (\pi_{A_i}(\mathbf{r}) - \pi_{B_j}(\mathbf{s})) \cup (\pi_{A_i}(\{\texttt{NEW}\}) - \pi_{B_j}(\mathbf{s})) \\ &= \emptyset \cup (\pi_{A_i}(\{\texttt{NEW}\}) - \pi_{B_j}(\mathbf{s})) \end{aligned} \tag{6.1}$$

Requiring Equation 6.1 to be equal to $\emptyset$ is equivalent to mandating that $\pi_{A_i}(\texttt{NEW}) \in \pi_{B_j}(\mathbf{s})$. This is the natural and efficient statement of the `TRIGGER`'s condition.

Now consider the impact of deleting a tuple `OLD` from $\boldsymbol{s}$, which replaces $\boldsymbol{s}$ by $\mathbf{s}' = \boldsymbol{s} - \{\texttt{OLD}\}$. It is not generally possible to distribute π in $\pi_{B_j}(\mathbf{s} - \{\texttt{OLD}\})$, since this

expression is $\pi_{B_j}(\mathbf{s})$ when $\texttt{OLD}.B_j$ occurs more than once in $\boldsymbol{s}$, and $\pi_{B_j}(\mathbf{s}) - \pi_{B_j}(\{\texttt{OLD}\})$ when $\texttt{OLD}.B_j$ occurs exactly once.

However, since B_j is a key for $\boldsymbol{s}$ (and thus $\texttt{OLD}.B_j$ occurs exactly once), $\pi_{B_j}(\mathbf{s} - \{\texttt{OLD}\})$ may be rewritten to $\pi_{B_j}(\mathbf{s}) - \pi_{B_j}(\{\texttt{OLD}\})$, and the constraint then becomes $\pi_{A_i}(\mathbf{r}) \subseteq (\pi_{B_j}(\mathbf{s}) - \pi_{B_j}(\{\texttt{OLD}\}))$. This is equivalent to $\pi_{A_i}(\mathbf{r}) \cap \pi_{B_j}(\{\texttt{OLD}\}) = \emptyset$. The reasoning of the preceding example may be generalized into the following fact.[4] We envision a library of such facts as part of a future implementation that leverages view-based constraints; see Chapter 7 for additional information.

Fact 6.2.1 (Set Containment involving Disjoint Sets)

Given sets S_1, S_2, and S_3 with $S_1 \subseteq S_2$.

$S_1 \cap S_3 = \emptyset \Leftrightarrow S_1 \subseteq (S_2 - S_3)$

There are two options for satisfying the foreign key constraint: not removing $\texttt{OLD}$ from $\boldsymbol{s}$ or removing every tuple $\texttt{OLD}_\mathbf{r}$ from $\boldsymbol{r}$ where $\pi_{A_i}(\{\texttt{OLD}_\mathbf{r}\}) = \pi_{B_j}(\{\texttt{OLD}\})$, *i.e.*, a cascading delete. In the former case, not performing an operation is always a possibility, as the operation's impact on the constraint would then be negated; this is handled in the `TRIGGER`'s action by terminating the operation. In the latter case, we essentially derive the `TRIGGER`'s action as well; namely, $\boldsymbol{r}$ becomes $\boldsymbol{r}' = \boldsymbol{r} - \{\texttt{OLD}_\mathbf{r}: \pi_{A_i}(\texttt{OLD}_\mathbf{r}) = \pi_{B_j}(\texttt{OLD})\}$.

[4] The use of *fact* denotes that the statement is well-understood and widely accepted.

More complex constraints, including in some cases those requiring ERA for their expression, can be handled similarly. We can extend strength reduction to more general cases as a result of the limited number of event types that we discussed earlier. Given a view-based constraint $V \subseteq W$ where $R \in sort(V)$, it is natural to consider $V \subseteq W$ as the `TRIGGER`'s condition in that the constraint must continue to hold even after the event occurs. Writing $V(\bullet)$ to indicate substitutions for R with instances for other relation schemas in V held fixed, $V(\mathbf{r})$ represents V before the trigger fires, and $V(\mathbf{r}')$ represents V as a result of the event occurring. In these cases, we assume that $R \notin sort(W)$. If we further restrict V to having only one instance of R in its expression and containing only the π, σ, $\bowtie$, $\cup$, and $\cap$ operations, we then may use strength reduction to address each event type. In fact, these restrictions – especially disallowing the set difference operation – allow us to focus only on the `INSERT` and `UPDATE` events, as a `DELETE` event on $\mathbf{r}$ does not impact the $V \subseteq W$ constraint under the current stipulations. That is, if a delete on $\mathbf{r}$ occurs, $V(\mathbf{r}') \subseteq V(\mathbf{r}) \subseteq W$ if V only contains π, σ, $\bowtie$, $\cup$, and $\cap$ operations. For the same reason, we only have to consider the aspect of an `UPDATE` event that concerns the addition of a new tuple, *i.e.*, the `INSERT` aspect. Thus, we combine the `INSERT` and `UPDATE` events into one case. It is worth noting that in some circumstances involving `UPDATE` events, we may be able to eliminate completely the need to check the constraint. For instance, there is no need to check $\pi_A(\texttt{NEW}) \in \pi_B(S)$ if $\pi_A(\texttt{OLD}) = \pi_A(\texttt{NEW})$, since this

implies membership in $\pi_B(S)$ by our precondition assumption.

We now present a series of facts and reductions in Reduction 6.2.2; these too will be part of the aforementioned implementation.

Reduction 6.2.2 (Core Relational Algebra Strength Reduction)
Based on the facts established in Table 6.1, the resulting strength reductions are captured in Table 6.2. For these facts and reductions, let $V \subseteq W$ be a view-based constraint where $R \notin sort(W)$ and V has only one instance of R in its expression and contains only the π, σ, $\bowtie$, $\cup$, and $\cap$ operations. In addition, $S \in sort(V)$, with $\mathbf{s}$ an instance of S. Enforcement of the constraint consists of performing the reduction on V and then determining if W contains the reduced result. The cases in table 6.2 represents the simple expressions involving individually the π, σ, $\bowtie$, $\cup$, and $\cap$ operations. More complicated expressions may be built from the composition of these operations and then reduced based on the cases in Table 6.2.

As for the more advanced ERA operations of importance (*i.e.*, aggregation and transitive closure), we can apply strength reduction in restricted cases involving aggregation on a single relation. Consider a scenario involving bank account balances and transactions that impact the balances. This example uses two tables: $ACCT$ and $TRANS$; their schemas are greatly simplified but nevertheless easily obtained by projections on tables from actual databases. The $ACCT$ table represents bank accounts with a column A_NUM containing account numbers and a column BAL

OPERATION	FACT
π	$\pi(\mathbf{r}') = \pi(\mathbf{r} \cup \{\texttt{NEW}\}) = \pi(\mathbf{r}) \cup \pi(\{\texttt{NEW}\})$
σ	$\sigma(\mathbf{r}') = \sigma(\mathbf{r} \cup \{\texttt{NEW}\}) = \sigma(\mathbf{r}) \cup \sigma(\{\texttt{NEW}\})$
$\bowtie$	$\mathbf{r}' \bowtie \mathbf{s} = \mathbf{s} \bowtie \mathbf{r}' = \mathbf{s} \bowtie (\mathbf{r} \cup \{\texttt{NEW}\}) = (\mathbf{s} \bowtie \mathbf{r}) \cup (\mathbf{s} \bowtie \{\texttt{NEW}\})$
$\cup$	$\mathbf{r}' \cup \mathbf{s} = \mathbf{s} \cup \mathbf{r}' = \mathbf{s} \cup (\mathbf{r} \cup \{\texttt{NEW}\}) = (\mathbf{s} \cup \mathbf{r}) \cup \{\texttt{NEW}\}$
$\cap$	$\mathbf{r}' \cap \mathbf{s} = \mathbf{s} \cap \mathbf{r}' = \mathbf{s} \cap (\mathbf{r} \cup \{\texttt{NEW}\}) = (\mathbf{s} \cap \mathbf{r}) \cup (\mathbf{s} \cap \{\texttt{NEW}\})$

Table 6.1: RA Facts Table

OPERATION	INSERT/UPDATE
π	$\pi(\{\texttt{NEW}\}) \subseteq W$
σ	$\sigma(\{\texttt{NEW}\}) \subseteq W$
$\bowtie$	$(\mathbf{s} \bowtie \{\texttt{NEW}\}) \subseteq W$
$\cup$	$\{\texttt{NEW}\} \subseteq W$
$\cap$	$(\mathbf{s} \cap \{\texttt{NEW}\}) \subseteq W$

Table 6.2: RA-Event Strength Reduction Table

containing account balances, and the $TRANS$ table represents bank transactions with a column T_NUM containing transaction numbers, a column A_NUM containing account numbers, and a column T_AMT containing transaction amounts. An expected constraint is that each account balance must match the sum of the transactions on the account and must be greater than or equal to zero. In Example 6.2.3, we provide a view-based ERA expression that enforces this constraint.

Example 6.2.3 (Account Balance Constraint)

$ACCT = \gamma_{A\_NUM,SUM(T\_AMT)\rightarrow BAL}(TRANS)$ AND $\sigma_{BAL<0}(ACCT) = \emptyset$

Using the convention employed earlier, consider $TRANS'$ as the table that results from inserting a tuple NEW into $TRANS$ and $ACCT'$ as the table that results from the

updating of the account balance in accordance with `NEW`. Momentarily ignoring the portion of the constraint that specifies $\sigma_{BAL<0}(ACCT) = \emptyset$, we have Example 6.2.3.1.

Example 6.2.3.1 (Post-`INSERT` Account Balance Constraint)

$ACCT' = \gamma_{A\_NUM,SUM(T\_AMT)\rightarrow BAL}(TRANS')$

To assist in the explication, we introduce Example 6.2.3.2 wherein we reconstitute the constraint into a form that uses aspects of relational calculus and the unnamed perspective, as discussed by Abiteboul, Hull, and Vianu in [AHV95]. In the following examples, we utilize the expression $\theta(v)$ to represent $\sigma_{A\_NUM=v}(TRANS)$ and $\theta'(v)$ to represent $\sigma_{A\_NUM=v}(TRANS')$.

Example 6.2.3.2 (Account Balance Constraint Restated)

Since A_NUM is the primary key for $ACCT$,

$$ACCT = \gamma_{A\_NUM,SUM(T\_AMT)\rightarrow BAL}(TRANS) \Leftrightarrow$$

$$\forall\ a \in ACCT.A\_NUM\ [ACCT(a,BAL) = (a, \sum_{t\in\theta(a)}\ t.T\_AMT)]$$

We can then express Example 6.2.3.1 as Example 6.2.3.3.

Example 6.2.3.3 (Post-`INSERT` Account Balance Constraint Restated)

$$\forall\ a \in ACCT'.A\_NUM\ [ACCT'(a,BAL) = (a, \sum_{t\in\theta'(a)}\ t.T\_AMT)]$$

As we presume that the constraint was true before we inserted `NEW`, we can then remove the unaffected portion of the constraint that does not concern the account

number present in NEW. Since the NEW pseudorecord functions as a variable, we can utilize it to compare the values in the new tuple to the values in the corresponding columns in the $ACCT$ and $TRANS$ tables. Furthermore, due to the distributive nature of addition, we can apply an additional optimization on the summation. This results in the expressions in Example 6.2.3.4.

Example 6.2.3.4 (Post-INSERT Account Balance Constraint Using NEW)

$$
\begin{aligned}
& ACCT'(\texttt{NEW}.A\_NUM, BAL) \\
&= (\texttt{NEW}.A\_NUM, \sum_{t \in \theta'(\texttt{NEW}.A\_NUM)} t.T\_AMT) \\
&= (\texttt{NEW}.A\_NUM, \sum_{t \in \theta(\texttt{NEW}.A\_NUM)} t.T\_AMT + \texttt{NEW}.T\_AMT)
\end{aligned}
$$

Naturally, we may express Example 6.2.3.4 algebraically, as we do in Example 6.2.3.5. The $\gamma_{SUM(T\_AMT)}\ (\sigma_{TRANS.A\_NUM=\texttt{NEW}.A\_NUM}(TRANS))$ expression in Example 6.2.3.5 is in effect the right-hand side of the first part of Example 6.2.3 constrained to only the rows in $TRANS$ that matches $\texttt{NEW}.A\_NUM$. Thus, we may replace it with $\pi_{BAL}(\sigma_{ACCT.A\_NUM=\texttt{NEW}.A\_NUM}\ (ACCT))$. This expression is of course the natural implementation for managing account balances; the significance lies in the fact that the implementation is formally deduced and thus proved correct by construction.

Example 6.2.3.5 (Rewritten Post-INSERT Account Balance Constraint)

$$\pi_{BAL}(\sigma_{ACCT'.A\_NUM=\texttt{NEW}.A\_NUM}\ (ACCT'))$$

$$= \gamma_{SUM(T\_AMT)}\ (\sigma_{TRANS.A\_NUM=\texttt{NEW}.A\_NUM}(TRANS)) + \pi_{T\_AMT}(\{\texttt{NEW}\})$$

$$= \pi_{BAL}(\sigma_{ACCT.A\_NUM=\texttt{NEW}.A\_NUM}(ACCT)) + \pi_{T\_AMT}(\{\texttt{NEW}\})$$

As a final step, consider that the right-hand side of Example 6.2.3.5 must be greater than or equal to zero (from $\sigma_{BAL<0}(ACCT) = \emptyset$). Leveraging Example 6.2.3.5, we arrive at Example 6.2.3.6.

Example 6.2.3.6 (Final Version of Account Balance Constraint)

$$\pi_{BAL}(\sigma_{ACCT.A\_NUM=\texttt{NEW}.A\_NUM}(ACCT)) \geq -\pi_{T\_AMT}(\{\texttt{NEW}\})$$

The reduction of Example 6.2.3 to Example 6.2.3.6 always applies because the grouping attribute is a primary key in the corresponding relation (*e.g.*, A_NUM is the primary key of $ACCT$). Example 6.2.3 demonstrates the capacity to apply strength reduction to aggregation involving summation on a single relation where the event type is an `INSERT`, and we can extend this application to include all aggregation functions that can distribute over the union of the original relation (that is, the relation before the event occurred) and the set containing the new tuple. These aggregation functions are `SUM`, `COUNT`, `MIN`, and `MAX`, and with each of them, we arrive at the same result by first evaluating the function with regard to the original relation and then factoring in the new tuple as we would if we evaluated the function using the relation that would result from the occurrence of event. We now present a

series of reductions in Reduction 6.2.4; these too will be part of the aforementioned implementation.

Reduction 6.2.4 (Aggregation Strength Reduction)

The strength reductions are captured in Tables 6.3, 6.4, and 6.5. For these reductions, given a view-based constraint $V \subseteq W$ and recalling Definition 4.3.8 from Chapter 4, let V be $\gamma_{\omega(A_i)\rightarrow\beta,X}(\mathbf{r})$ where $\{A_i\} \cup X \subseteq sort(R)$, $A_i \notin X$, and $\omega \in \{$`SUM`, `COUNT`, `MIN`, `MAX`$\}$. In addition, let $\Phi = \sigma_\theta(\gamma_{\omega(A_i)\rightarrow\beta,X}(\mathbf{r}))$ where θ is defined in the following for each event. Enforcement of the constraint consists of performing the reduction on V and then determining if W contains the reduced result. As we saw in Example 6.2.3.6, additional reductions may be possible. In addition, notice that some cases conditionally have a reduction; if the condition is not met, then no reduction is possible.

`INSERT` **event:** In the case of the `INSERT` event, let θ select the tuple from V whose values for all attributes from X match those from `NEW` (that is, `NEW`'s attribute values from X).

`DELETE` **event:** In the case of the `DELETE` event, let θ select the tuple from V whose values for all attributes from X match those from `OLD` (that is, `OLD`'s attribute values from X).

`UPDATE` **event (1):** In the case of the `UPDATE` event where `OLD` and `NEW` match on X

(denoted $\sigma_X(\texttt{OLD}) = \sigma_X(\texttt{NEW})$), let θ select the tuple from V whose values for all attributes from X match those from OLD (that is, OLD's attribute values from X).

UPDATE event (2): In the case of the UPDATE event where OLD and NEW do not match on X (denoted $\sigma_X(\texttt{OLD}) \neq \sigma_X(\texttt{NEW})$), reduce to the INSERT and DELETE cases for the appropriate θs.

	AGGREGATION	
EVENT	SUM	COUNT
INSERT	$\Phi.\beta + \texttt{NEW}.A_i$	$\Phi.\beta + 1$
DELETE	$\Phi.\beta - \texttt{OLD}.A_i$	$\Phi.\beta - 1$
UPDATE (1)	$\Phi.\beta + (\texttt{NEW}.A_i - \texttt{OLD}.A_i)$	$\Phi.\beta$

Table 6.3: Event-Aggregation (SUM and COUNT) Strength Reduction Table

EVENT	MIN
INSERT	$\texttt{NEW}.A_i \leq \Phi.\beta \Rightarrow \texttt{NEW}.A_i$ $\texttt{NEW}.A_i > \Phi.\beta \Rightarrow \Phi.\beta$
DELETE	$\texttt{OLD}.A_i \neq \Phi.\beta \Rightarrow \Phi.\beta$
UPDATE (1)	$\texttt{NEW}.A_i \leq \Phi.\beta \Rightarrow \texttt{NEW}.A_i$ $\texttt{NEW}.A_i > \Phi.\beta \wedge \Phi.\beta \neq \texttt{OLD}.A_i \Rightarrow \Phi.\beta$

Table 6.4: Event-Aggregation Strength (MIN) Reduction Table

EVENT	MAX
INSERT	$\texttt{NEW}.A_i \geq \Phi.\beta \Rightarrow \texttt{NEW}.A_i$ $\texttt{NEW}.A_i < \Phi.\beta \Rightarrow \Phi.\beta$
DELETE	$\texttt{OLD}.A_i \neq \Phi.\beta \Rightarrow \Phi.\beta$
UPDATE (1)	$\texttt{NEW}.A_i \geq \Phi.\beta \Rightarrow \texttt{NEW}.A_i$ $\texttt{NEW}.A_i < \Phi.\beta \wedge \Phi.\beta \neq \texttt{OLD}.A_i \Rightarrow \Phi.\beta$

Table 6.5: Event-Aggregation Strength (MAX) Reduction Table

In this section, we have illustrated the capacity of the duality to serve as a tool that enables the application of strength reduction to the definition of triggers. In the examples that we discuss, the perspective granted by the duality helps to provide simple, computationally inexpensive conditions for use in the trigger definitions. These conditions contrast sharply with the ones that a naive approach likely suggests. For instance, in Example 6.2.3.1, computing each side of the expression and then comparing the resulting sets is a simple method for determining the satisfaction of the constraint; however, this expensive computation is not necessary, as we have shown.

Furthermore, consider the approach possibly taken if the only specification provided were the prose description. In this scenario, one may have chosen more procedurally oriented techniques that use *for loops* and the like. However, the practice of programming involves trial and error, and in the interest of delivering the desired functionality correctly and expediently, it is preferable to precisely describe the desired results and have the correct implementation derived.

An additional observation concerns the global nature of views, as discussed in Chapters 1 and 2. Throughout this section, we obviously use the aspect of the duality that concerns the use of views in expressing constraints, and thus we have defined the constraints at a global level. However, as we have seen in cases involving insertions, deletions, and updates, the enforcement of the constraints often entail items at the local level, *i.e.*, tuples. This discrepancy between global and local actually highlights

a potential benefit of using views to express constraints.

To illustrate this point, consider the examples from earlier in this section. In both examples, we isolate the local element – the newly inserted, deleted, or updated tuples – from the globally defined constraint (that is, a constraint expressed using views), and based on an assertion that the constraint held before the action (the insertion, deletion, or update), we then remove the original constraint from the expression. As a result, we only have to examine the newly inserted, deleted, or updated tuples and not the entire relation, and in each case, this follows quickly from defining the constraint globally. We thus observe the value of defining globally and enforcing locally.

6.3 Tree Conformity

In this section, we investigate the impact of view-based constraints on *tree conformity*. As the name suggests, a *tree conformity* constraint occurs when the data encodes two trees that must have the same structure. The most natural example of *tree conformity* concerns the relationship between the specification of parts and their subparts and the actual part instances. As the specification of parts and their subparts involves the establishment of hierarchical relationships between parts and their subparts, we can characterize these relationships as parent-child relationships and thus represent the relationships as a tree with the product (*e.g.*, truck) serving

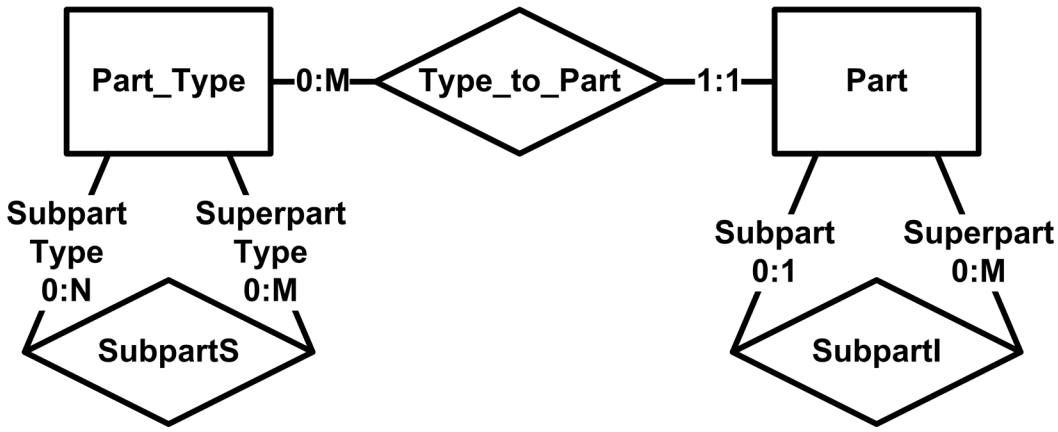

Figure 6.1: ER Diagram for Tree Conformity

as the root. Since each product instance should follow its specification's hierarchy, we may represent the product instances as trees as well. Ensuring that product instances meet the product specification then becomes a matter of ensuring that the instance trees conform to the specification tree; we denote this as an example of the tree conformity constraint.

To assist in this explication, we reprint Figure 1.1 from Chapter 1 in Figure 6.1. From this figure, a natural expression of an example of the tree conformity constraint is Example 6.3.1; recall that α is the transitive closure operation.

Example 6.3.1 (Specification-Instance Tree Conformity Constraint Example)

$Type\_to\_Part \circ \alpha(SubpartI) \subseteq \alpha(SubpartS) \circ Type\_to\_Part$

To understand the constraint specified in Example 6.3.1, imagine beginning with

some root part type τ, and then construct the *type* tree rooted at τ using $\alpha(SubpartS)$. In addition, find all corresponding parts of type τ using the $Type\_to\_Part$ relation, make certain that these parts are roots (*i.e.*, finished products), and then construct an instance tree from each of these roots using $\alpha(SubpartI)$. Then the structure of each of the instance trees must match the specification tree's structure, with the $Type\_to\_Part$ relation providing a mapping from type to part that enables the verification of the matching. The constraint specified in Example 6.3.1 satisfies this imperative.

While Example 6.3.1 does specify an example of the tree conformity constraint, it also involves transitive closure on each of the expression, and from a computational perspective, it is certainly better to write the constraint without transitive closure if that is possible. This leads to Theorem 6.3.2.

Theorem 6.3.2 (Tree Conformity Theorem)

Let A, B, and C be binary relations over $X \times Y$, $Y \times Y$, and $X \times X$, respectively where A is a function from Y into X. If $A \circ B \subseteq C \circ A$, then $A \circ \alpha(B) \subseteq \alpha(C) \circ A$.

Proof:

By the definition of transitive closure involving finite binary relations, both $\alpha(B)$ and $\alpha(C)$ must eventually reach fixpoints where their cardinalities do not increase. Let n be the maximum of the number of compositions that $\alpha(B)$ performs to reach its fixpoint and the number of compositions that $\alpha(C)$ performs

to reach its fixpoint. In the following, we utilize B^i to represent $\underbrace{B \circ \cdots \circ B}_{i}$ and C^j to represent $\underbrace{C \circ \cdots \circ C}_{j}$. Thus, $B^n = \alpha(B)$ and $C^n = \alpha(C)$ by the choice of n.

Basis: $A \circ B \subseteq C \circ A$

This is the theorem hypothesis.

Induction: If $A \circ B^{n-1} \subseteq C^{n-1} \circ A$, then $A \circ B^n \subseteq C^n \circ A$

By the definition of relational composition, $A \circ B^n = A \circ (B^{n-1} \circ B)$.

Since relational composition is associative, $A \circ (B^{n-1} \circ B) = (A \circ B^{n-1}) \circ B$.

By the inductive hypothesis, $A \circ B^{n-1} \subseteq C^{n-1} \circ A$, and thus $(A \circ B^{n-1}) \circ B \subseteq (C^{n-1} \circ A) \circ B$.

By associativity, $(C^{n-1} \circ A) \circ B = C^{n-1} \circ (A \circ B)$, and using the Basis, $C^{n-1} \circ (A \circ B) \subseteq C^{n-1} \circ (C \circ A)$.

Since relational composition is associative, $C^{n-1} \circ (C \circ A) = (C^{n-1} \circ C) \circ A$.

By the definition of relational composition, $(C^{n-1} \circ C) \circ A = C^n \circ A$.

Therefore, we have the following:

$$\begin{aligned} A \circ B^n &= A \circ (B^{n-1} \circ B) \\ &= (A \circ B^{n-1}) \circ B \\ &\subseteq (C^{n-1} \circ A) \circ B \\ &\subseteq C^{n-1} \circ (A \circ B) \\ &\subseteq C^{n-1} \circ (C \circ A) \\ &\subseteq (C^{n-1} \circ C) \circ A \\ &\subseteq C^n \circ A \end{aligned}$$

As a result, $A \circ B^n \subseteq C^n \circ A$. □

We can then use Theorem 6.3.2 to simplify Example 6.3.1 to Example 6.3.3. That is, if Example 6.3.3 is true, then by Theorem 6.3.2, Example 6.3.1 must be true.

Example 6.3.3 (Final Version of Specification-Instance Tree Conformity Constraint Example)

$Type\_to\_Part \circ SubpartI \subseteq SubpartS \circ Type\_to\_Part$

The simplification found in Example 6.3.3 is an artifact of using the view-based constraint approach to specify constraints.

6.4 Chapter Summary

In this chapter, we investigate the second hypothesis in the database systems domain. We demonstrate the usefulness of view-based constraints in recovering constraint violators. In addition, we establish the manner in which view-based constraints enable the application of strength reduction as well as the simplification of complex expressions that involve tree conformity.

7

Conclusion and Future Work

In this chapter, we begin with a summary of the previous chapters. We then present the major results from this work and plans for future work.

7.1 Summary

In this section, we summarize the previous chapters. We begin in Chapter 1 where we present a motivating example and an outline for the remainder of the document.. In Chapter 2, we next discuss the necessary background information and introduce the notational conventions that we employ. Included in these topics are the Zachman Framework for Enterprise Architecture, the core relational model, relational algebra, relational calculus, and Structured Query Language (SQL) as well as a discussion of related work.

In Chapter 3, we then discuss the concepts foundational to our hypotheses and examine the difficulties that we encounter in establishing the duality. We also prove the duality's existence in a formal context (FOL) and then introduce contexts that provide empirical evidence for the duality, using the Zachman Framework for Enterprise Architecture to organize and guide in this exploration. Our hypotheses validation process tends to focus on the Specify and Design roles, as the Build role emphasizes the implementation-related representations that fall outside our purview.

In Chapter 4, we begin the process of finding empirical evidence of the duality by closely inspecting the database systems domain (in the Zachman Framework, the What column). We focus, in this chapter, on specific instances of the duality in the database systems domain. In particular, we concentrate on the well-known constraints and dependencies found in the core relational model, utilizing relational algebra and set operations to illustrate the manner in which this facet of the duality holds. We also describe a set of extensions (including aggregate functions, transitive closure, and metadata operations) to the core relational model and investigate related classes of constraints.

We then leave the data models and proceed to address models and methodologies at the conceptual level that are less formal and more descriptive. Here, we identify two archetypal elements that allow us to argue the existence of the duality in other areas of the Zachman Framework and thus to continue our hypothesis validation process.

We continue our examination by delving into the duality in an area that we entitle cross-column modeling.

After we validate the first hypothesis, we address the use of the duality as a tool in the development of information systems in Chapter 6. Specifically, we demonstrate instances in the context of relational database management systems where the duality has great utility.

7.2 Results

In this section, we provide a synopsis of the major results from this research. Please recall our hypotheses stated in Chapters 1 and revisited in Chapter 3:

1. **a view↔constraint duality exists in database systems and systems engineering domains and**

2. **the view↔constraint duality is a conceptually tractable and useful tool in the development of information systems.**

In the following, we divide our results into those associated with the first hypothesis and those associated with the second hypothesis.

First Hypothesis: Existence of View↔Constraint Duality

For purposes of demonstrating that the first hypothesis holds, we employ two techniques. The first technique is the canonical direct proof, and in Theorem 3.2.1, we formally establish the view↔constraint duality in the context of FOL. The other technique is empirical evidence, and we utilize this approach in the relational database domain to discuss instances of views expressing constraints. In this domain, though, we are still able to leverage the relational model and thus express the evidence in a formal manner using relational algebra and relational calculus. We augment our presentation of the duality in the relational database domain by extending the core relational algebra to include operators for transitive closure, aggregation, and metadata and demonstrating the manner in which views express constraints in these extensions.

We continue to employ the empirical evidence technique as we investigate instances of the duality outside the relational database domain. As we conduct this investigation in the more abstract and less formal Specify and Design roles of the Zachman Framework, we are driven to use examples to help indicate the manner in which the duality holds in these other areas, and to lend structure to the discussion, we utilize concepts such as relations, relational operations, and arrays.

As we see throughout the process of investigating the first hypothesis, the accuracy with which the hypothesis holds depends upon the degree to which the constraints

can be formalized. In the case of the relational data model, the constraints are well-understood and can be formalized. This is even true of the *semantic* constraints that are expressible in relational algebra and relational calculus. In the case of the other models, the constraints from the Specify and Design roles have a less formal expression, and views prove to be a method for interjecting more formality into the expression of the constraints.

Second Hypothesis: Usefulness of View↔Constraint Duality

To demonstrate that the second hypothesis holds, we again turn toward the relational model to exhibit examples of the duality having practical uses. In Theorem 6.1.4, we see the manner in which view-based constraints may be effectively rewritten to remove ambiguity and indeterminism in the constraints and thereby to identify the violators of the constraints. We also document the method by which view-based constraints enable the application of strength reduction to the enforcement of the constraints. Moreover, we prove in Theorem 6.3.2 the capacity of view-based *tree conformity* constraints to be transformed into much simpler expressions that avoid potentially computationally expensive transitive closure operations.

As the uses of the duality that we investigate are entirely in the relational model domain, this suggests that our comments regarding the accuracy of the first hypothesis also hold for the second hypothesis; that is, the accuracy with which the second

hypothesis holds depends upon the degree to which the constraints can be formalized. In the case of the relational data model, the constraints that we explore in Chapter 6 can be formalized, and thus we can generate general solutions for them. Uses of the duality outside the relational domain is a focus of future work. We also observe that views furnish a method for specifying constraints at a *global* level instead of a *local* (that is, instance) level, and this is very useful when constraints must hold across all possible instances. There is also a psychological benefit. Specifically, users often have difficulty with quantifiers, but the global nature of view-based constraints may make them more conceptually tractable.

7.3 Future Work

In this section, we discuss future work that leverages the aspects of the duality that we have discussed throughout this document. This work primarily focuses on using view-based constraints.

View-based Constraint Implementation

Our plans include an implementation that natively supports view-based constraints in relational databases, as discussed in Chapter 6. As stated in Chapter 2, currently the SQL `ASSERTION` as outlined in [Ame99] is not widely implemented by

the database management system vendors, but in our implementation, an open source database management system will be extended to natively support view-based constraint expressions via the SQL `ASSERTION` statement.

This implementation will also contain the library of facts and reductions that we present in Chapter 6. These optimizations in conjunction with the use of the database management system's existing query optimizer will result in optimized expressions that enforce the semantic constraints specified by the user. These optimized expressions may prove helpful in the typically query processing performed by the DBMS, and thus the implementation may involve an investigation of the database management system's query processing mechanism to ascertain whether that mechanism would benefit from utilizing the optimized view-based constraint.

Visual Specification

In addition to providing a means to implement the view-based constraints within the DBMS, it is equally important to create an effective method for users to express the view-based constraints. Our plans involve the development of a user interface that enables the user to visually specify constraints on models; this user interface will be in the spirit of the extensions to the standard ER notation that we see in Figures 4.1 and 4.2.

In light of our hypotheses, this user interface will be founded on a facility to

specify and connect SQL `VIEW`s visually. This will allow us to target application domains where the view-based constraints may be particularly useful and serve to guide discovery. Furthermore, this enables us to leverage view-based constraints to effectively manage aggregation and transitive closure, as the handling of aggregation and transitive closure is a major challenge in existing solutions.

Analysis and Design Aid

To build on the planned implementation efforts discussed in the preceding, we endeavor to suggest a new perspective on information systems analysis and design that leverages the capacity of views to express constraints. The key idea that view-based constraints enable users to utilize a familiar yet precise medium to express business rules. As we have discussed previously, a view is generally an external representation or schema of some underlying structure or entity; examples include relational database `VIEW`s, object views, reports, XML files, text files (extracted from another source), and user interfaces.

In short, views allow the user to see a familiar, consistent "face" while permitting the designer or developer to change the internal representation. Naturally, this does require a mapping from the view to the underlying model, but as we have seen, this already exists in many cases. Furthermore, this has the potential for application in a wide range of areas. For instance, one may envision this approach as being an

important tool in the migration to a new system from a legacy system. In this context, the users likely already have a great familiarity with the legacy user interface, and as this interface provides an operational view of data and processes, the designer or developer may utilize it with the users to determine the organization's business rules. Another context is in the area of computer forensics where the well-known security benefits of relational database `VIEW`s may prove helpful in forensic investigations; an exploration of the application of view-based constraints to evidentiary specificity is forthcoming.

The evaluation of this new perspective begins with the aforementioned implementation efforts. The outcomes of these efforts will be instructive as to the viability of the view-based constraints as a general information systems analysis and design tool.

Bibliography

[AHV95] S. Abiteboul, R. Hull, and V. Vianu. *Foundations of Databases*. Addison-Wesley, New York, 1995.

[Alb03] Stephen Albin. *The Art of Software Architecture: Design Methods and Techniques*. John Wiley & Sons, Inc., New York, NY, USA, 2003.

[Ame92] American National Standards Institute. *American National Standard for Information Systems: Database Language – SQL: ANSI X3.135-1992*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1992.

[Ame99] American National Standards Institute. *American National Standard for Information Systems: Database Language – SQL: ANSI X3.135-1999*. American National Standards Institute, 1430 Broadway, New York, NY 10018, USA, 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Princiles, Techniques, and Tools.* Addison-Wesley, 1986.

[AU94] Alfred V. Aho and Jeffrey D. Ullman. *Foundations of Computer Science.* W. H. Freeman & Co., New York, NY, USA, 1994.

[BBC80] Philip A. Bernstein, Barbara T. Blaustein, and Edmund M. Clarke. Fast maintenance of semantic integrity assertions using redundant aggregate data. In *VLDB*, pages 126–136. IEEE Computer Society, 1980.

[BC79] Peter Buneman and Eric K. Clemons. Efficient monitoring relational databases. *ACM Trans. Database Syst.*, 4(3):368–382, 1979.

[BLT86] José A. Blakeley, Per-Åke Larson, and Frank Wm. Tompa. Efficiently updating materialized views. In Carlo Zaniolo, editor, *SIGMOD Conference*, pages 61–71. ACM Press, 1986.

[BT05] Bernhard Beckert and Kerry Trentelman. Second-order principles in specification languages for object-oriented programs. In Geoff Sutcliffe and Andrei Voronkov, editors, *LPAR*, volume 3835 of *Lecture Notes in Computer Science*, pages 154–168. Springer, 2005.

[CB01] Thomas M. Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[CB04] James F. Cox and John H. Blackstone, editors. *APICS Dictionary*. American Production and Inventory Control Society, 11th edition, 2004.

[CFPT94] Stefano Ceri, Piero Fraternali, Stefano Paraboschi, and Letizia Tanca. Automatic generation of production rules for integrity maintenance. *ACM Trans. Database Syst.*, 19(3):367–422, 1994.

[CGL07] Stefano Ceri, Francesco Di Giunta, and Pier Luca Lanzi. Mining constraint violations. *ACM Trans. Database Syst.*, 32(1):6, 2007.

[CM93] Mariano P. Consens and Alberto O. Mendelzon. Low complexity aggregation in graphlog and datalog. *Theoretical Computer Science*, 116(1&2):95–116, 1993.

[Cod70] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, 1970.

[Cod72] E. F. Codd. Relational completeness of data base sublanguages. *In: R. Rustin (ed.): Database Systems: 65-98, Prentice Hall and IBM Research Report RJ 987, San Jose, California*, 1972.

[Cor] General Motors Corporation. Follow a Truck Down the Assembly Line. http://www.gm.com/company/gmability/edu_k-12/9-12/making_vehicles/manufacturing/index.html. visited 4/23/2006.

[CTF88] Marco A. Casanova, Luiz Tucherman, and Antonio L. Furtado. Enforcing inclusion dependencies and referencial integrity. In Francois Bancilhon and David J. DeWitt, editors, *VLDB*, pages 38–49. Morgan Kaufmann, 1988.

[CW90] Stefano Ceri and Jennifer Widom. Deriving production rules for constraint maintainance. In Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek, editors, *VLDB*, pages 566–577. Morgan Kaufmann, 1990.

[CW91] Stefano Ceri and Jennifer Widom. Deriving production rules for incremental view maintenance. In Guy M. Lohman, Amílcar Sernadas, and Rafael Camps, editors, *VLDB*, pages 577–589. Morgan Kaufmann, 1991.

[CW94] Stefano Ceri and Jennifer Widom. Deriving incremental production rules for deductive data. *Inf. Syst.*, 19(6):467–490, 1994.

[DB82] Umeshwar Dayal and Philip A. Bernstein. On the correct translation of update operations on relational views. *ACM Trans. Database Syst.*, 7(3):381–416, 1982.

[Del78] Claude Delobel. Normalization and hierarchical dependencies in the relational data model. *ACM Trans. Database Syst.*, 3(3):201–222, 1978.

[DeM79] Tom DeMarco. *Structured Analysis and System Specification*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1979.

[EN06] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems (5th Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[Fag77] Ronald Fagin. Multivalued dependencies and a new normal form for relational databases. *ACM Trans. Database Syst.*, 2(3):262–278, 1977.

[FC85] Antonio L. Furtado and Marco A. Casanova. Updating relational views. In *Query Processing in Database Systems*, pages 127–142. Springer, 1985.

[fFA] Zachman Institute for Framework Advancement. Zachman Framework Overview. http://www.zifa.com/framework.html. visited 4/27/2006.

[FS00] Martin Fowler and Kendall Scott. *UML distilled (2nd ed.): a brief guide to the standard object modeling language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[FSdS79] Antonio L. Furtado, Kenneth C. Sevcik, and Clesio Saraiva dos Santos. Permitting updates through views of data bases. *Inf. Syst.*, 4(4):269–283, 1979.

[GGGM98] Parke Godfrey, John Grant, Jarek Gryz, and Jack Minker. Integrity constraints: Semantics and applications. In Jan Chomicki and Gunter Saake, editors, *Logics for Databases and Information Systems*, pages 265–306. Kluwer, 1998.

[GHQ95] Ashish Gupta, Venky Harinarayan, and Dallan Quass. Aggregate-query processing in data warehousing environments. In Umeshwar Dayal, Peter M. D. Gray, and Shojiro Nishio, editors, *VLDB*, pages 358–369. Morgan Kaufmann, 1995.

[GKM92] Ashish Gupta, Dinesh Katiyar, and Inderpal Singh Mumick. Counting solutions to the view maintenance problem. In *Workshop on Deductive Databases, JICSLP*, pages 185–194, 1992.

[GM96] Erich Grädel and Gregory L. McColm. Hierarchies in transitive closure logic, stratified datalog and infinitary logic. *Annals of Pure and Applied Logic*, 77(2):169–199, 1996.

[GMUW02] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database Systems: The Complete Book.* Prentice Hall, Upper Saddle River, NJ, 2002.

[HMN82] Lawrence J. Henschen, William McCune, and Shamim A. Naqvi. Compiling constraint-checking programs from first-order formulas. In *Advances in Data Base Theory*, pages 145–169, 1982.

[IEE00] IEEE. *IEEE Std 1471-2000, IEEE Recommended Practice for Architectural Descriptions of Software Intensive Systems.* Institute of Electrical and Electronics Engineers, New York, NY, USA, 2000.

[Imm87] Neil Immerman. Languages that capture complexity classes. *SIAM Journal on Computing*, 16(4):760–778, 1987.

[ISO00] ISO. *ISO 14258: Industrial automation systems – Concepts and rules for enterprise models.* International Organization for Standardization, Geneva, Switzerland, 2000.

[IZG97] William H. Inmon, John A. Zachman, and Jonathan G. Geiger. *Data Stores, Data Warehousing and the Zachman Framework: Managing Enterprise Knowledge.* McGraw-Hill, Inc., New York, NY, USA, 1997.

[Kel86] Arthur M. Keller. Unifying database and programming language concepts using the object model. In Klaus R. Dittrich and Umeshwar Dayal, editors, *OODBS*, pages 221–222. IEEE Computer Society, 1986.

[Koc05] Ned Kock. *Business Process Improvement Through E-Collaboration: Knowledge Sharing Through the Use of Virtual Groups.* Idea Group Publishing, 2005.

[KP81] Shaye Koenig and Robert Paige. A transformational framework for the automatic control of derived data. In *VLDB*, pages 306–318. IEEE Computer Society, 1981.

[Lib01] Leonid Libkin. Expressive power of SQL. *Lecture Notes in Computer Science*, 1973:1–21, 2001.

[LS06] James J. Lu and M. Jeremy Scoggins. Specifying and solving boolean constraint problems in relational databases: a case study. In *ACM-SE 44: Proceedings of the 44th annual Southeast regional conference*, pages 399–404, New York, NY, USA, 2006. ACM Press.

[LV03] Jens Lechtenbörger and Gottfried Vossen. On the computation of relational view complements. *ACM Trans. Database Syst.*, 28(2):175–208, 2003.

[MC94] Thomas W. Malone and Kevin Crowston. The interdisciplinary study of coordination. *ACM Computing Survey*, 26(1):87–119, 1994.

[Mor83] Matthew Morgenstern. Active databases as a paradigm for enhanced computing environments. In Mario Schkolnick and Costantino Thanos, editors, *VLDB*, pages 34–42. Morgan Kaufmann, 1983.

[OMG05] OMG. *Object Constraint Language Specification, version 2.0.* Object Modeling Group, June 2005.

[Pai82] Robert Paige. Applications of finite differencing to database integrity control and query/transaction optimization. In *Advances in Data Base Theory*, pages 171–209, 1982.

[PJ00] Meilir Page-Jones. *Fundamentals of object-oriented design in UML.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[Por06] Victor Portougal. *Cases on Information Technology: Lessons Learned, Volume 7*, chapter XXIII: ERP Implementation for Production Planning at EA Cakes Ltd. Idea Group Publishing, 2006.

[Pre04] Roger S Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Science/Engineering/Math, 2004.

[PS05] Victor Portougal and David Sundaram. *Business Processes: Operational Solutions for SAP Implementation*. Idea Group Publishing, 2005.

[QW91] Xiaolei Qian and Gio Wiederhold. Incremental recomputation of active relational expressions. *IEEE Trans. Knowl. Data Eng.*, 3(3):337–341, 1991.

[RBIM98] Dave Roberts, Dick Berry, Scott Isensee, and John Mullaly. *Designing for the User with OVID: Bridging the Gap Between Software Engineering and User Interface Design*. Macmillan Technical Publishing, 1998.

[RG00] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems*. McGraw-Hill, Boston, 2000.

[RS79] Lawrence A. Rowe and Kurt A. Shoens. Data abstractions, views and updates in rigel. In Philip A. Bernstein, editor, *SIGMOD Conference*, pages 71–81. ACM, 1979.

[RSS96] Kenneth A. Ross, Divesh Srivastava, and S. Sudarshan. Materialized view maintenance and integrity constraint checking: Trading space for time. In H. V. Jagadish and Inderpal Singh Mumick, editors, *SIGMOD Conference*, pages 447–458. ACM Press, 1996.

[RT06] Roberta Russell and Bernard W. Taylor. *Operations Management: Quality and Competitiveness in a Global Environment.* John Wiley & Sons, Inc., 5th edition, 2006.

[Sof] Popkin Software. Popkin Enterprise Architecture Framework. http://government.popkin.com/frameworks/zachman_framework.htm. visited 4/23/2006.

[SZ92] J. F. Sowa and J. A. Zachman. Extending and formalizing the framework for information systems architecture. *IBM Systems Journal*, 31(3), 1992. IBM Publication G321-5488.

[Tar41] Alfred Tarski. On the calculus of relations. *The Journal of Symbolic Logic*, 6(3):73–89, 1941.

[TG01] Can Türker and Michael Gertz. Semantic integrity support in SQL:1999 and commercial (object-)relational database management systems. *VLDB J.*, 10(4):241–269, 2001.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, volume Volume I - Fundamental Concepts. Computer Science Press, New York, 1988.

[UM04] Scott Urman and Lisa McClain. *Oracle Database 10g PL/SQL Programming.* McGraw-Hill Osborne Media, 2004.

[WBD03] Jeffrey L. Whitten, Lonnie D. Bentley, and Kevin C. Dittman. *Systems Analysis and Design.* McGraw-Hill, Inc., New York, NY, USA, 2003.

[WK03] Jos Warmer and Anneke Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[WM91] Paul T. Ward and Stephen J. Mellor. *Structured Development for Real-Time Systems.* Prentice Hall Professional Technical Reference, 1991.

[WR05] Catharine M. Wyss and Edward L. Robertson. A formal characterization of PIVOT/UNPIVOT. In *CIKM '05: Proceedings of the 14th ACM International Conference on Information and Knowledge Management*, pages 602–608, New York, NY, USA, 2005. ACM Press.

[Wre98] Robert Wrembel. Object-oriented views: Virtues and limitations. In *13th International Symposium on Computer and Information Sciences - ISCIS98*, Antalya, November 1998.

[Zac87] J. A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 26(3), 1987. IBM Publication G321-5298.

[Zac99] John A. Zachman. A framework for information systems architecture. *IBM Systems Journal*, 38(2/3):454–470, 1999.

[Zan76] Carlo Zaniolo. Analysis and design of relational schemata for database systems. 1976.

[ZBC02] Richard Zanibbi, Dorothea Blostein, and James R. Cordy. Recognizing mathematical expressions using tree transformation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 24(11):1455–1467, 2002.

Curriculum Vitae

JOHN A. SPRINGER

401 North Grant Street
West Lafayette, IN 47907
(765) 494-2560
E-mail: jospring@cs.indiana.edu

EDUCATION

Ph.D., Computer Science, Indiana University, Bloomington, IN, USA, August 2007
M.S., Computer Science, Indiana University, Bloomington, IN, USA, June 2002
B.S., Mathematics/Systems Analysis, Taylor University, Upland, IN, USA, May 1995

TEACHING EXPERIENCE

2006-Present	Assistant Professor	Purdue University
1999-2004	Associate Instructor	Indiana University

RESEARCH INTERESTS

Constraints
Views
Query Languages
Knowledge Representation and Management

PROFESSIONAL EXPERIENCE

2004-2006	Implementation Engineer	Bostech Corporation
2002-2004	Independent consultant	
1999	President	Infinite Loop Consulting
1998	Systems Analyst	Eli Lilly and Company
1995-1998	Consultant	Andersen Consulting

PROFESSIONAL AFFILIATIONS

Association for Computing Machinery (ACM)
Institute of Electrical and Electronics Engineers (IEEE)

PUBLICATIONS

Refereed journals

Martin, R., Robertson, E., and Springer, J. (2005). Architectural Principles for Enterprise Frameworks: Guidance for Interoperability. In P. Bernus and M. Fox (Eds.), Knowledge Sharing in the Integrated Enterprise: Interoperability Strategies for the Enterprise Architect (pp. 79-91). Boston, MA: Springer Boston.

Refereed conferences and workshops

Martin, R., Robertson, E., and Springer, J. (2004). Architectural Principles for Enterprise Frameworks. In CAiSE 2004 Workshops in connection with The 16th Conference on Advanced Information Systems Engineering (pp. 151-162). Riga, Latvia: Faculty of Computer Science and Information Technology, Riga Technical University.

Springer, J. (2003). An Exploration of Enterprise Architecture Framework Views. In Proceedings of the 1st International Conference on Enterprise Information Systems (ICEIS) Doctoral Consortium (DCEIS-2003) (pp. 46-48). Angers, France: ICEIS Press.

Technical reports

Martin, R., Robertson, E., and Springer, J. (2004). Architectural Principles for Enterprise Frameworks. Indiana University Computer Science Department Technical Report 594.