An approach to developing complex database schemas using form types

Ivan Luković1,∗,†, Pavle Mogin2, Jelena Pavićević3 and Sonja Ristić4

1University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovića 6, 21000 Novi Sad, Serbia
2Victoria University of Wellington, P.O. Box 600, Wellington, New Zealand
3Crnogorski Telekom and University of Montenegro, Moskovska 29, 81000 Podgorica, Montenegro
4University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovića 6, 21000 Novi Sad, Serbia

SUMMARY

In this paper we consider an approach to developing complex database schemas. Apart from the theoretical model of the approach, we also developed a CASE tool named Integrated Information Systems*Case, R.6.2 (IIS*Case) that supports the practical application of the approach. In this paper the basis of our approach to the design and integration of database schemas and ways of using IIS*Case is outlined. The main features of a new version of IIS*Case, developed in Java, are described. IIS*Case is based on the concept of ‘form type’ and supports the conceptual modelling of a database schema, generating subschemas and integrating them into a relational database schema in 3NF. IIS*Case provides an intelligent support for complex and highly formalized design and programming tasks. Having an advanced knowledge of information systems and database design is not a compulsory prerequisite for using IIS*Case. IIS*Case is based on a methodology of gradual integration of independently designed subschemas into a database schema. The process of independent subschema design may lead to collisions in expressing real-world constraints. IIS*Case uses specialized algorithms for checking the consistency of constraints embedded in a database schema and its subschemas. This paper briefly outlines the application of the process of detecting collisions, and actions the designer may take to resolve them. Copyright © 2007 John Wiley & Sons, Ltd.

Received 7 November 2005; Revised 5 February 2007; Accepted 11 February 2007

KEY WORDS: database schema design and integration; subschema; form type; CASE tool; formal consistency; IIS*Case; agile software development

*Correspondence to: Ivan Luković, University of Novi Sad, Faculty of Technical Sciences, Trg D. Obradovića 6, 21000 Novi Sad, Serbia.
†E-mail: ivan@uns.ns.ac.yu

Copyright © 2007 John Wiley & Sons, Ltd.
INTRODUCTION

Conceptual database (DB) schema modelling is mainly based on the Entity-Relationship (ER) model or Unified Modelling Language (UML) class diagrams. Today, many software tools, which rely on these techniques, support (i) the conceptual design of DB schemas, (ii) transformation of conceptual DB schemas into implementation (mainly relational) DB schemas, and (iii) their implementation under different database management systems (DBMSs).

In general, there are two basic approaches to a DB schema design: (a) the direct approach, and (b) the approach of a gradual integration of external schemas. In the direct approach, user requirements are analyzed all at once and the whole DB schema is created directly. This approach may be appropriate in the case of a small DB schema design, but it is inappropriate in the case of a complex DB schema design. The second approach is used when the number and complexity of user requirements exceeds a designer’s power of perception. Design of a complex DB schema is based on a gradual integration of external schemas. An external schema is a structure that formally specifies a user’s view of a DB schema at the conceptual level. Each transaction program that supports a user request is based on an external schema. After their creation, external schemas are integrated into a conceptual DB schema.

Using design methodologies based on the second approach and techniques such as ER modelling or UML, and even the relational data model and an appropriate CASE tool, requires advanced knowledge, skills, and high perception power. Failing to find an appropriate number of designers that possess these properties may lead to a risk of designing a poor quality DB schema.

Besides, these methods and techniques are often incomprehensible to end-users. In practice, that may lead to problems in communication and to misunderstandings between designers and end-users. As a rule, misunderstandings result in a poorly designed DB schema, because support of all the specified user requirements is not ensured. Usually, both designers and end-users become aware of that too late, when the DB schema is already implemented.

Therefore, it is a challenge to provide an alternative approach, and a CASE tool, which support an automated DB schema design that it is based on concepts end-users are familiar with. A designer who understands and follows the rules for creating design specifications imposed by such a tool would be able to design DB schemas more quickly and easily, even if their complexity extends beyond the limits of usual human perception.

The aim of this paper is to present such an alternative approach and a CASE tool that supports its practical application. We believe that it may help in resolving or at least alleviating the aforementioned problems. The approach is tested in practice by applying it in a number of projects, in which the authors took part as project team members.

In this paper the fundamentals of the methodological approach to design and integration of DB schemas that is based on the form type concept is outlined. We also present ways of using IIS*Case, a tool that we developed to support the approach. The main features and functionality of a new version of Integrated Information Systems*Case, R.6.2 (IIS*Case) are also described. Despite the fact that [1–4] present ideas that are to a certain extent similar to ours, to the best of our knowledge, the approach and the CASE tool presented here cannot be found in the same form elsewhere in the literature.

IIS*Case is designed to provide complete support for developing DB schemas which are complex with regard to the number of concepts used, and to give intelligent support during that process. IIS*Case supports:
• conceptual modelling of a DB schema;
• automated design of relational database subschemas in the 3rd normal form;
• checking the consistency of constraints embedded into a DB schema and a set of subschemas;
• automated integration of subschemas into a relational DB schema; and
• generating an SQL/DDL schema specification.

The remainder of the paper is organized as follows. We introduce the main assumptions related to the process of conceptual DB schema modelling and integration. Then, we define the concept of the form type and, after that, we focus our attention on the design and integration of DB schemas. First, we consider the process of generating a single subschema, and then the process of detecting and resolving collisions between the generated subschemas and their integration into a unified relational DB schema. Then, we give an example of applying IIS*Case in the process of a DB schema design. Finally, we discuss some results of applying the approach and the tool in industry.

RELATED WORK

Approaches to DB schema design based on the concept of form type have a long tradition. A number of authors, as in [2,3], introduced the concept to make the DB design process more comprehensible to end-users. However, all these works were focused on creating procedures for generating DB schema in the ER data model using the specifications of form types. In our approach, we assume that the form type concept may be used for conceptual DB schema design instead of the ER data model. Thus, we focus on creating procedures for generating relational DB schemas using specifications of form types. In this way, we widely utilize the powerful mathematical formalisms that the relational data model is based on. Our main idea stems from the late 1980s (see, e.g., [5–12]). Although there are references that consider generating ER DB schemas from form types and there is an approach [1] that uses form types and relies on the universal relation scheme assumption, we did not find any references covering all aspects of our approach.

A model-driven tool for agile software development, named DeKlarit™ is presented in [1]. The business component is the main modelling concept of DeKlarit. A business component is an unnormalized hierarchical structure comprising attributes and key constraints. It represents a view of a future relational DB schema. According to the universal relation scheme assumption, each attribute in DeKlarit is identified only by its name. A hierarchical structure of a business component bears information about functional dependencies. Therefore, DeKlarit provides the automatic generation of a relational DB schema by the synthesis algorithm, whose input is a set of functional dependencies inferred from all the business components. After generating a relational DB schema, DeKlarit generates appropriate SQL/DDL scripts for different DBMSs. It supports generating not only first-cut SQL/DDL scripts, but also SQL/DDL and SQL/DML scripts for regenerating an existing DB schema and its current data. For each business component, DeKlarit can generate a transaction program. There are many similarities between the approach of DeKlarit and our approach in IIS*Case, but there are also significant differences. Our form type concept corresponds to the business component of DeKlarit. We also utilize the synthesis algorithm for relational DB schema design. However, unlike DeKlarit, IIS*Case can generate not only relation scheme keys and basic referential integrity constraints, but also unique constraints and other more complex interrelation constraints. Also, IIS*Case supports the
concept of an application system that allows the decomposition of a large DB schema design into a number of smaller subschema designs that can be implemented independently and concurrently. Then, IIS*Case provides algorithms for integrating independently designed subschemas into a unified DB schema. These algorithms rely on the detection of collisions of constraints embedded into different subschemas.

In Gálvez et al. [4], the authors present methods for the analysis and design of cooperative object-oriented information systems. A model of an information system is split into several subsystems that can be processed more easily. Interrelationships existing between such subsystems require that the development methods are applied cooperatively. The authors introduce the following cooperative tools, based on a powerful and user-friendly graphical interface and working over a Cooperative Data Dictionary (CDD): CDFF—a cooperative data flow diagram tool; CDB—a tool for cooperatively designing the database; CSCR—a tool for designing the visual appearance of the application; CREP—a tool for generating reports; and CGT—a code generator. The main focus of the paper is on CDB. CDB incorporates SMF and FMS. SMF is a semantic model whose basic component is the form, i.e. the form structure, which the authors call design. It supports modelling entities, their attributes/fields, and relationships. FMS is a Form Management System, whose task is to store designs and verifying their consistency. CDB is basically a user-interface that allows the creation of designs. The concept of the form in [4] is very similar to our concept of the form type. However, the IIS*Case form type may provide more information about various types of relational DB schema constraints. Furthermore, it carries additional information concerning the embedded functionality of future transaction programs made over such a form type. It will support the implementation of a code generator for application prototypes, executable in different programming environments.

Since designs can be made by different users, conflicts may arise. Gálvez et al. [4] distinguish structural and semantic incompatibilities between designs. Structural conflicts arise when the same attribute is included in different schemas with different type and format definitions. Semantic conflicts are caused by homonyms and synonyms. Detecting conflicts is possible by a conflict management system in CDB, and it takes place at the level of the designs created. IIS*Case provides more powerful consistency control at the level of generated relational subschemas that should be integrated into a unified relational DB schema. Apart from detecting homonyms, the process of consistency checking identifies collisions for various constraint types. It is even able to automatically resolve some of them.

The term ‘data integration’ is used differently in various references. For example, it is often used to denote methods and techniques for selecting portions of existing data from various data sources in order to display them, or store them in a new database. In these applications, the main motivation is to solve the problem of accessing heterogeneous data sources in a unified manner, and the problem of integrating DB schemas is usually not considered. All such approaches are outside the scope of our paper.

Our approach to database integration is different. We apply methods and techniques to integrate relational subschemas into a unified DB schema. The main motivation of our approach is to help designers in a cooperative development of a highly complex DB schema. Rahm and Bernstein [13], generally consider the problem of data integration through a survey of approaches to schema matching. They address various DB application domains for schema matching, such as (i) DB schema integration, (ii) an extracting, transforming, and loading (ETL) process in a data warehouse, (iii) exchanging messages (i.e. data) in e-commerce systems, and (iv) semantic query processing. The main motivation of our approach coincides with the definition of the goal of DB schema integration in Section 2.1 ‘Schema integration’ of [13].

Copyright © 2007 John Wiley & Sons, Ltd.

DOI: 10.1002/spe
One of the uses of the term ‘data integration’ is to denote the integration of existing databases and their schemas. A typical example is data re-engineering in legacy information systems. If the schemas of different data sources are expressed in different data models, they need to be transformed into a selected data model before the integration. Alagic [14] introduces a formalism named database institution, which is ‘capable of handling data models with significantly different structural properties’. The main goal is to provide such schema transformations from one data model to another one that preserve both structural properties and integrity constraints of a DB schema. Schmalz et al. [15] consider the problem of integrating heterogeneous legacy databases. They introduce a formalism that provides a way for logical integration of heterogeneous databases, called EITH. EITH supports a unified representation of a DB schema and application code. Furthermore, it supports translation of DB schemas expressed in various data models (particularly ER) into a unified abstract representation. Thus, a comparison of various schema and code constructs may be obtained. The scope of their paper is much wider than ours, because we consider neither schema transformations nor the integration of existing legacy data. On the other hand, our approach considers the formal consistency of relational DB schema constraints in detail. Since subschemas of our approach may also represent different data sources of a legacy system, the reports on the constraint collisions detected by IIS*Case may be helpful in designing procedures for restructuring and integrating existing data.

Lawrence and Barker [16,17] consider the problem of integrating relational DB schemas using a standardized dictionary. According to their approach, schema integration is a process of combining multiple database schemas into a coherent integrated view. Their integration architecture constructs an integrated view automatically, by combining local views. This approach is similar to ours, in which relational subschemas represent local views that are integrated into a unified DB schema. In Lawrence and Barker [16,17] local views are defined by independently created XML documents that express database semantics. In our approach, external views are expressed by form types. Each form type is also a tree structure that IIS*Case can automatically transform into an XML document. In Lawrence and Barker [16,17] the system maps user queries expressed using the high-level concepts to schema elements representing underlying data sources. The architecture proposed there is a query-only system and database updates are not considered. In contrast, our approach and IIS*Case support the design of a unified DB schema and subschemas that are aimed not only for queries, but also for safe updates that guarantee DB consistency.

CONCEPTUAL MODELLING AND INTEGRATION OF A DATABASE SCHEMA

The design of a complex DB schema is based on a gradual integration of external schemas. In contrast to some other approaches, in our approach and IIS*Case, external schemas are expressed by sets of form types. The notion of the form type is defined in the following section. The main motivation of introducing and using the form type is that this concept is closer to the end-users’ perception of data, than it would be, for example, to the concepts of entity and relationship types in the ER data model. On the other hand, the form type is a concept that is formal enough to allow precise expression of all of the rules significant for structuring a DB schema.

The integration of a DB schema is done at the implementation level instead of the conceptual level. A DB schema at the implementation level is expressed by the relational data model. In contrast to the ER model or UML, the relational data model offers much wider possibilities to formalize and
automate the process of detecting collisions between external schemas and integrating external schemas into a unified DB schema. That statement arises from the fact that the relational model combined with the predicate calculus offers a very suitable and flexible mathematical formalism for identifying, classifying and specifying various database constraint types. Formal definitions of various constraint types in relational data model may be found in Lukovic et al. [18] and Mogin et al. [19], while the theoretical foundations of the DB schema integration process, based on a rigorous mathematical approach, are presented in Lukovic et al. [6] and Ristic et al. [20]. Also, we combine the form type approach with the relational data model in the DB schema design. The form type data model is semantically rich enough for a successful use in specifying conceptual DB schemas of business information systems. On the other hand, unlike the ER data model, the form type data model is relatively simple, and consequently, much closer to the end-users.

Since the integration of a DB schema is done at the implementation level, the notion of a subschema is introduced. A subschema is obtained by expressing an external schema by concepts of the relational data model [21]. A DB schema is obtained by gradual integration of subschemas.

THE FORM TYPE

The form type is the central concept of our approach and IIS*Case. This concept is presented in, for example, [7–9]. A form type generalizes document types, i.e. screen forms that users utilize to communicate with an information system. IIS*Case imposes strict structuring rules on form types. Using this tool, a designer specifies screen forms of transaction programs and, indirectly, creates an initial set of attributes and constraints. During the automated process of the design, which demands a minimal interaction with the designer, the form types are transformed into an implementation DB schema.

A form type is a named tree structure, whose nodes are called component types. Each component type is identified by its name within the scope of a form type, and has non-empty sets of attributes and keys, and a possibly empty set of unique constraints. A set of allowed database operations must be associated with each component type. The set of allowed operations is a subset of the set of ‘standard’ operations {retrieve, insert, update, delete}. If the update operation is associated with a component type, the set of updatable attributes of the component type must be also specified. In addition, each attribute of a component type may be mandatory or optional. If a component type attribute must not have null values, it is designated as mandatory. Otherwise, it is optional.

Each attribute of a component type is selected from a global set of all information system attributes. The attributes are globally identified only by their names. IIS*Case imposes strict rules for specifying attributes and their domains, and for specifying component type attributes. Specifications of domains, attributes and component type attributes in the IIS*Case environment may be found in [8,10,18,21,22].

*Example 1.* Figure 1(b) is a simplified representation of the structure of a form type F, which generalizes the screen form in Figure 1(a). A screen for specifying form types in IIS*Case is presented in Figure 1(c). The form type consists of two component types, STUDENT and GRADES, which are graphically represented by rectangles. The sets of component type attributes are also shown. Each solid line in Figure 1(b) underlines all the attributes of a component type key, whereas each dashed line underlines all the attributes of a unique constraint. A small rectangle near the upper-right corner of a component type represents allowed operations that are associated with the component type,
where $r$ stands for retrieve, $i$ for insert, $u$ for update, and $d$ for delete. Grade is the only updatable attribute of the component type GRADES (see the field ‘Update allowed’ in Figure 1(c)).

Each form type has instances. A form type instance represents a particular document that a user creates, modifies or uses to communicate with the information system. In our approach, a form type instance is a tree structure over the instances of component types. An instance of the root component type is, at the same time, an instance of a form type itself. It is identified by the value of any of the root component type keys. If a component type $N_i$ is directly superordinated to the component type $N_j$, then each (child) instance of $N_j$ is associated with exactly one (parent) instance of $N_i$ and each instance of $N_i$ may be related to more than one instance of $N_j$. Each instance of $N_j$ is identified by a value of a key of $N_j$, but only within the scope of its parent instance.
Example 2. Each instance of the component type STUDENT in Figure 1(b) represents a particular screen form aimed at browsing and editing records about exams passed by a student. Figure 2 shows two possible instances of the form type in Figure 1(b).

Each instance of the component type STUDENT is identified by a value of the key StudentId. Each instance of the component type GRADES represents an exam passed by a student. Each student may have passed many exams, or may not have any, and each passed exam is always associated with exactly one student. Each passed exam is identified by a value of the key CourseId within the scope of the value of the parent component type key StudentId. Date is a unique constraint of the component type GRADES, which means that each non-null value of Date must be unique among all GRADES instances of a student identified by a value of StudentId.

The form type structuring rules provide automatic inference of relational DB constraints from form types. These constraints are processed by the synthesis algorithm later in the process of a DB schema design, which is discussed in the following section. The initial set of constraints, inferred from a form type, consists of:

- a set of functional dependencies \( F \);
- a set of non-functional dependencies \( NF \);
- a set of special functional dependencies \( Fu \); and
- a set of null value constraints \( Nc \).

The sets \( F \) and \( NF \) are defined in [8,9,23]. The set \( F \) contains all non-trivial functional dependencies, and the set \( NF \) contains all trivial functional dependencies, for which only a ‘many-to-many’ relationship between attributes on the left-hand side is semantically relevant, and there is no real attribute in the set of all attributes which is on the right-hand side of the functional dependency. For example, suppose we are interested only in the relationship between students and courses, but not for any property of their relationship. Then, the left-hand side of the non-functional dependency would be \{StudentId, CourseId\}, while the right-hand side would be a dummy attribute \( \theta \). The notion of a set of special functional dependencies \( Fu \) is defined in Pavicevic [10]. \( Fu \) is inferred from unique constraints defined by form types. These three sets of dependencies are processed by the synthesis algorithm. In that process, special functional dependencies are treated differently. Their left-hand sides
are not built into keys, as it is the case with the set $F \cup NF$, but into the unique constraints of the generated relation schemes.

A set of null value constraints contains a clause for each form type attribute. That clause specifies whether the attribute may have null values or not. Null value constraints of the form type attributes are mapped into null value constraints of the corresponding relation schema attributes in the process of generating a relational DB schema.

**Example 3.** From the form type $F$ in Example 1, one can infer the following sets of constraints:

- $F(F) = \{\text{StudentId} \rightarrow \text{Year}, \text{StudentId}+\text{CourseId} \rightarrow \text{Date}, \text{StudentId}+\text{CourseId} \rightarrow \text{Grade}\}$;
- $F_{\emptyset}(F) = \{\text{StudentId}+\text{Date} \rightarrow \text{CourseId}\}$;
- $NF(F) = \{\text{StudentId}+\text{CourseId} \rightarrow \theta\}$; and
- $N_c(F) = \{\text{Null}(F, \text{StudentId}) = \perp, \text{Null}(F, \text{Year}) = \perp, \text{Null}(F, \text{CourseId}) = \perp, \text{Null}(F, \text{Date}) = \top, \text{Null}(F, \text{Grade}) = \perp\}$,

where the symbol $\rightarrow$ denotes a special functional dependency inferred from a unique constraint of a component type, and $\theta$ denotes a semantically non-interpreted attribute, which will not appear in a DB schema. In this particular example, the non-functional dependency $\text{StudentId}+\text{CourseId} \rightarrow \theta$ is covered by the functional dependency $\text{StudentId}+\text{CourseId} \rightarrow \text{Date}$, and it is therefore logically redundant. However, there are many examples, in which non-functional dependencies are not redundant, and consequently must be considered in the process of a DB schema design. $\text{Null}(F, A) = \perp$ denotes that null values for $A$ are not allowed in the form type, whereas $\text{Null}(F, A) = \top$ has the opposite meaning (see the field ‘Mandatory’ of the screen form in Figure 1(c)).

**DATABASE SCHEMA GENERATING AND INTEGRATING**

A DB schema design in the IIS*Case environment is organized by decomposing the corresponding information system into application systems [8,10,11]. An *application system* is a specification of a subsystem of a future information system. The set of all application systems of an information system is organized as a tree structure. It is the *application system tree* of the information system. Thus, each application system may include one or more child application systems. All child application systems of an application system are its *application subsystems*. Figure 3 depicts two different application system trees in IIS*Case: ‘Faculty Organization’ and ‘Billing System’.

Each form type in IIS*Case is designed in the context of an application system. Therefore, a set of form types is a part of an application system, and is an input specification for the process of the DB schema design. The design process generates new specifications, which become components of the application system.

A DB schema design in the IIS*Case environment is an iterative process that includes the following steps:

1. conceptual modelling; and
2. integrating a relational DB schema.

Conceptual modelling is performed by creating sets of form types, one for each application system. The next step is performed using the results of the conceptual modelling. As a recursive process,
Figure 3. Project trees in IIS*Case.

it starts with leaf application systems and ends with the root application system. It includes the following steps:

2.1. declaring subschemas;
2.2. generating a potential DB schema/subschema; and
2.3. consolidation of the potential DB schema and its subschemas.

The steps above are performed for each application system. Step 2.1 declares a subschema for each directly subordinated application subsystem of the selected application system. Step 2.2 generates a relational DB schema for the selected application system. It is called a potential DB schema. At the next level of recursion, a potential subschema may be declared as a subschema. Detecting and resolving collisions between a potential DB schema and its subschemas is performed in Step 2.3.
Figure 4. Steps of the DB schema design process.

Figure 4 depicts the steps of the DB schema design process, where dotted arrows represent data flows and solid arrows represent process flows. After conceptual modelling, Step 2.1 starts for each leaf application system. Since leaf application systems do not have any application subsystems, Step 2.1 completes trivially, without declaring any subschema. Step 2.2 generates a potential DB schema for a selected leaf application system. Step 2.3 is also trivial and completes successfully. Therefore, a potential DB schema becomes a DB schema for the selected application system. Let us consider a non-leaf application system, at the next level of recursion. In Step 2.1, for each directly subordinated application subsystem of the selected application system, the previously generated
(potential) DB schema is declared as a subschema. Step 2.2 generates a potential DB schema of the application system. In Step 2.3, collisions between the potential DB schema and its subschemas are detected and resolved. There are two possible outcomes of the collision detection process. (i) If collisions are detected, the process must be interrupted and the collisions have to be resolved. Collisions are commonly resolved by returning to Step 1. After resolving collisions, the integration process should be restarted. (ii) Otherwise, the potential DB schema becomes the DB schema of the considered application system, and the process continues by selecting the next application system and returning to Step 2.1. When the root application system is successfully processed, the overall process is complete. The potential DB schema of the root application system becomes the final DB schema of the information system.

Generating a relational DB schema (i.e. a potential DB schema/subschema) is described in the following section, while the consolidation process is described in the section ‘Consolidation’.

IIS*Case allows for a designer to review and validate results obtained in each step of the design process. For example, a designer may review the generated relation schemes and constraints of a potential DB schema, and check their compatibility with the corresponding concepts of subschemas. If a designer is not satisfied with generated results, or there are some incompatibilities, they can go back one or more steps, change form types or relation schemes, and repeat the process. Designers are allowed to make only those changes to relation schemes that are invariant to the conceptual design. Examples of these changes include proclaiming another candidate key as a primary key, or proclaiming a key as a key in the scope of a subschema [6].

Generating a relational database schema

The process of generating a relational DB schema of an application system in the IIS*Case environment consists of the following five steps:

- generating a set of relation schemes by the synthesis algorithm;
- generating the closure graph;
- detecting candidates for primary keys;
- propagating primary keys; and
- generating interrelation constraints.

All the steps are executed in the specified order, on a designer’s request. The input specification is the union of the sets of form types of a selected application system and all its application subsystems. Figure 5 shows the IIS*Case screen form that a designer uses to control the process of the relational DB schema generation. The form on the left-hand side represents the case when none of the specified steps has been completed. A designer selects the steps that will be executed using the corresponding check boxes. If they select any item except the first, all the preceding items are selected automatically. The form on the right-hand side represents the case when all the specified steps are completed. A tick mark next to an item indicates that the corresponding step has been successfully completed.

The first step is applying the synthesis algorithm [23,24]. During this step the conceptual design is transformed into the relational data model. The transformation starts by inferring the sets of functional, non-functional, and special functional dependencies from all the form types included in the input specification and results in a set of relation schemes with sets of attributes, keys, and unique constraints [8–10,18]. The synthesis algorithm of Beeri and Bernstein [23] is extended in IIS*Case by a
new step that generates both synthesized and non-synthesized keys [8]. The second step generates the closure graph that presents the generated schema graphically. Informally, each node of the closure graph represents a relation scheme, generated by the synthesis algorithm and each directed edge represents the fact that a proper or improper subset of a key of a subordinated node is propagated as a foreign key into a superior node. The notion of the closure graph is introduced and presented in [10,19]. In the third step, from the set of all equivalent keys of each relation scheme, candidates for a primary key are automatically identified. In the fourth step, the designer declares one of the candidate keys (if there is more than one) as the primary key of a relation scheme, and IIS*Case propagates it automatically as a foreign key in all directly superordinated relation schemes. Propagating also removes the keys that are equivalent to the propagated one from relation schemes in superior nodes.

The current version of IIS*Case generates a ‘pure’ relational DB schema, where primary keys are propagated as foreign keys in superior relation schemes. Whenever there are at least two primary key candidates in a relation scheme, IIS*Case gives the freedom to a designer to decide which one to select as the primary key. A future version of IIS*Case will also support the generation of an object-relational DB schema. In such a schema, the role of a primary key for each relation scheme inevitably takes an object identifier, which is also propagated further as a foreign key.

The fifth step generates the interrelation constraints of the following types [8,10,19,25,26]:

- basic referential integrity constraints;
- extended referential integrity constraints;
- referential integrity constraints based on non-trivial inclusion dependencies; and
- inverse referential integrity constraints, both basic and those based on non-trivial inclusion dependencies.

Basic and extended referential integrity constraints are inferred from the closure graph [10,25]. Their existence is a consequence of the primary key propagation. If the whole primary key is transferred from a subordinated (referenced) relation scheme to the superior (referencing) one,
IIS*Case generates a basic referential integrity constraint. Otherwise, it detects an extended referential constraint. The extended referential integrity constraint is introduced in [19].

Referential integrity constraints based on non-trivial inclusion dependencies arise from the non-trivial inclusion dependencies that a designer may define at the level of the set of all information system attributes. Inverse referential integrity constraints arise from the form type components. More precisely, if a component type \( N_i \) is directly superordinated to a component type \( N_j \), then a designer may declare that each instance of \( N_i \) must be related to at least one instance of \( N_j \). In that case, IIS*Case infers an inverse referential integrity constraint.

Every interrelation constraint of any of the mentioned types is represented by an edge in the closure graph. Different types of referential integrity constraints have different edge colours.

IIS*Case also detects the existence of homonyms and A-dependent or B-dependent relation schemes in the same step. These terms are discussed in [19,27].

In this way, a relational DB schema, i.e. a potential DB schema of a selected application system, is generated. If the application system is, at the same time, a subsystem of another application system, then the generated DB schema becomes a subschema.

**Consolidation**

The process of independent design of external schemas may lead to collisions in expressing the real-world constraints and business rules. The process of consolidation and integration of a number of external schemas expressed by form types would be hard to formalize, since the form type model does not possess precisely defined constraint inference rules. On the other hand, in the relational data model, inference rules and the corresponding algorithms for its main constraint types are available. This makes it feasible to specify formally and automate the process of consolidation of relational schemas.

The integration process is not a mere unifying of subschemas. Our approach to the integration is based on detecting and resolving constraint collisions that may arise among a potential DB schema and its subschemas of an application system. If collisions are detected, at least one subschema is formally not consistent with the potential database schema. Programs made over inconsistent subschemas do not guarantee safe database updates. Detecting and resolving constraint collisions are accomplished in Step 2.3, ‘Consolidation of the potential DB schema and its subschemas’.

The integration process may only go on from Step 2.3 to Step 2.1 if all the subschemas contain compatible sets of constraints. Colliding constraints may be embedded into subschemas for various reasons, but the main one is independent modelling of their form types. Thus, appropriate collision resolving procedures must be applied. The process of consolidation is defined in [6,12,20]. The steps of the consolidation process are organized by constraint types, and they check:

- consistency of attribute sets of a subschema relation schemes and a potential DB schema;
- consistency of sets of key constraints;
- consistency of sets of unique constraints;
- consistency of sets of null value constraints; and
- consistency of sets of referential integrity constraints.

These steps are always executed sequentially, in the specified order. Each step in the list above cannot be executed before all the collisions of the previous step have been resolved. If for all subschemas the constraint sets of all analyzed types are consistent with the constraint sets of a potential DB schema,
DEVELOPING COMPLEX DATABASE SCHEMAS USING FORM TYPES

that potential DB schema is adopted as the DB schema of an application system. If the application system is the root application system, the DB schema becomes the final DB schema of the information system.

The consistency checking of a potential database schema and its subschemas is initiated by selecting the IIS*Case option DB Schema Analysis, which is given in Figure 6. The form on the left-hand side represents the case in which none of the specified steps have been completed. A designer selects the step that will be executed using the corresponding check boxes. If he or she selects any item except the first, all the preceding items are selected automatically. The form on the right-hand side represents the case when the first step has been completed with collisions detected. The following steps cannot be executed, unless all the collisions are resolved. All the possible collisions are described in detail in the appropriate IIS*Case reports. The reports also contain instructions and explanations for their interpretation.

AN EXAMPLE OF A DB SCHEMA GENERATION

A very simplified and hypothetical model of a university information system consists of two application systems, Student Service and Faculty Organization, where Student Service is the only application subsystem of Faculty Organization. These application systems contain the form types shown in Figures 7 and 8, respectively.

Figure 7 shows the only form type defined in Student Service, which refers to information about student’s grades (STG). Figure 8 shows form types of the application system Faculty Organization. These form types contain information about the faculty organization units (FCU), employees as persons (PER), courses and examinations (CEX), and students (STU). A dot appearing at the component type
FACULTY within the form type FCU denotes that each instance of FACULTY must be related to at least one instance of DEPARTMENT. Analogously, a dot at the component type DEPARTMENT denotes that each instance of DEPARTMENT must be related to at least one instance of MAJOR.

A designer introduced the attributes Dean, Lecturer, and Examiner by renaming the attribute PersonId, while the attribute Prerequisite was introduced by renaming CourseId. Renaming is an action of introducing a new attribute that refers to an existing attribute. A renamed attribute has to be domain compatible with the one to which it refers. It is introduced to specialize the meaning of the referred attribute, and consequently to avoid homonyms in a DB schema. For example, the renamed attribute Lecturer denotes PersonIds of persons who are lecturers.

IIS*Case defines the following non-trivial inclusion dependencies \([\text{Dean}] \subseteq [\text{PersonId}], [\text{Lecturer}] \subseteq [\text{PersonId}], [\text{Examiner}] \subseteq [\text{PersonId}], \) and \([\text{Prerequisite}] \subseteq [\text{CourseId}]\) at the level of the set of all information system attributes, as an automatic result of renaming.

We shall demonstrate the main idea of our approach to DB schema design in the IIS*Case environment using the example described above.

**Generating a potential DB schema and a subschema**

The conceptual DB schema of the application system Faculty Organization includes the specifications of form types created in both the application system Faculty Organization and its subsystem Student Service, whereas the conceptual DB schema of the application system Student Service includes only its own form types.

The process of integrating the relational DB schema (Figure 4, Step 2) starts with the only leaf application system Student Service. Step 2.1, declaring subschemas, is trivial. Step 2.2, applying the algorithms for generating a potential DB schema (Figure 5), results in sets of relation schemes and interrelation constraints. Relation schemes are structured as three-tuples whose components are a set of attributes, a set of keys, and a set of unique constraints. Primary keys are underlined.
Figure 8. The form types of the application system Faculty Organization.
The set of relation schemes of the application system *Student Service* is:

- \( \text{Student}([\text{StudentId}, \text{Year}], \{\text{StudentId}\}, \}) \); and
- \( \text{Grade}([\text{StudentId}, \text{CourseShortName}, \text{Date}, \text{Grade}], \{\text{StudentId}+\text{CourseShortName}\}, \}) \).

The set of generated basic referential integrity constraints of the application system *Student Service* is:

- \( \text{RI}_\text{Grade}_\text{Student}: \text{Grade}([\text{StudentId}] \subseteq \text{Student}([\text{StudentId}]) \).

Step 2.3, ‘Consolidation’, is also trivial for the leaf application system *Student Service*. In this way, the first iteration of the integrating process produces the *Student Service* DB schema.

The next iteration of the integration process starts by selecting the parent application system *Faculty Organization*. In step 2.1, the DB schema of the child application system *Student Service* is declared as the only subschema of *Faculty Organization*. Step 2.2 produces a potential DB schema of *Faculty Organization*.

The resulting set of relation schemes after step 2.2 is:

- \( \text{Faculty}([\text{FacId}, \text{FacShortName}, \text{FacName}, \text{Dean}], \{\text{FacId}, \text{FacShortName}, \text{FacName}\}, \{\text{Dean}\}) \);
- \( \text{Department}([\text{FacId}, \text{DepId}, \text{DepName}], \{\text{FacId}+\text{DepId}, \text{FacId}+\text{DepName}\}, \}) \);
- \( \text{Major}([\text{FacId}, \text{DepId}, \text{MajId}, \text{MajName}], \{\text{FacId}+\text{DepId}+\text{MajId}, \text{FacId}+\text{DepId}+\text{MajName}\}, \}) \);
- \( \text{Person}([\text{PersonId}, \text{Name}, \text{Addr}, \text{Phone}, \text{Email}], \{\text{PersonId}\}, \{\text{Email}\}) \);
- \( \text{Course}([\text{CourseId}, \text{CourseShortName}, \text{CourseName}, \text{Prerequisite}, \text{Lecturer}, \text{Year}, \text{Semester}], \{\text{CourseId}, \text{CourseShortName}\}, \}) \);
- \( \text{Timetable}([\text{CourseId}, \text{Day}, \text{Time}, \text{Classroom}], \{\text{CourseId}+\text{Day}+\text{Time}\}, \}) \);
- \( \text{Exam}([\text{CourseId}, \text{ExamId}, \text{DateTime}, \text{Examiner}], \{\text{CourseId}+\text{ExamId}\}, \}) \);
- \( \text{Student}([\text{StudentId}, \text{PersonId}, \text{FacId}, \text{DepId}, \text{Year}], \{\text{StudentId}\}, \}) \);
- \( \text{Major}_\text{Student}([\text{StudentId}, \text{FacId}, \text{DepId}, \text{MajId}], \{\text{StudentId}+\text{FacId}+\text{DepId}+\text{MajId}\}, \}) \); and
- \( \text{Grade}([\text{StudentId}, \text{CourseId}, \text{Grade}, \text{Date}], \{\text{StudentId}+\text{CourseId}\}, \}) \).

The set of basic referential integrity constraints is:

- \( \text{RI}_\text{Department}_\text{Faculty}: \text{Department}([\text{FacId}] \subseteq \text{Faculty}([\text{FacId}]) \);
- \( \text{RI}_\text{Major}_\text{Department}: \text{Major}([\text{FacId}, \text{DepId}]) \subseteq \text{Department}([\text{FacId}, \text{DepId}]) \);
- \( \text{RI}_\text{Timetable}_\text{Course}: \text{Timetable}([\text{CourseId}]) \subseteq \text{Course}([\text{CourseId}]) \);
- \( \text{RI}_\text{Exam}_\text{Course}: \text{Exam}([\text{CourseId}]) \subseteq \text{Course}([\text{CourseId}]) \);
- \( \text{RI}_\text{Grade}_\text{Course}: \text{Grade}([\text{CourseId}]) \subseteq \text{Course}([\text{CourseId}]) \);
- \( \text{RI}_\text{Student}_\text{Person}: \text{Student}([\text{PersonId}]) \subseteq \text{Person}([\text{PersonId}]) \);
- \( \text{RI}_\text{Student}_\text{Department}: \text{Student}([\text{FacId}, \text{DepId}]) \subseteq \text{Department}([\text{FacId}, \text{DepId}]) \);
- \( \text{RI}_\text{Grade}_\text{Student}: \text{Grade}([\text{StudentId}]) \subseteq \text{Student}([\text{StudentId}]) \);
- \( \text{RI}_\text{Major}_\text{Student}_\text{Major}: \text{Major}_\text{Student}([\text{StudentId}]) \subseteq \text{Student}([\text{StudentId}]) \); and
- \( \text{RI}_\text{Major}_\text{Student}_\text{Major}: \text{Major}_\text{Student}([\text{FacId}, \text{DepId}, \text{MajId}]) \subseteq \text{Major}([\text{FacId}, \text{DepId}, \text{MajId}]) \).
The set of referential integrity constraints inferred from non-trivial inclusion dependencies is:

- \( R_{JC\_Faculty\_Person}: Faculty[Dean] \subseteq Person[PersonId]; \)
- \( R_{JC\_Course\_Person}: Course[Lecturer] \subseteq Person[PersonId]; \)
- \( R_{JC\_Exam\_Person}: Exam[Examiner] \subseteq Person[PersonId]; \) and
- \( R_{JC\_Course\_Course}: Course[Prerequisite] \subseteq Course[CourseId]. \)

The set of inverse referential integrity constraints is:

- \( IRI\_Faculty\_Department: Faculty[FacId] \subseteq Department[FacId]; \) and
- \( IRI\_Department\_Major: Department[(FacId, DepId)] \subseteq Major[(FacId, DepId)]. \)

The last step of generating a potential DB schema (Figure 5) includes detecting potential homonyms and DB schema dependencies.

In the case study, IIS*Case detects the existence of A-dependent relation schemes [27] and potential homonyms, and generates a DB Schema Dependency and Homonym Diagnostic Report (Figure 9).

Relation schemes Course and Student are A-dependent owing to the existence of a common referencing relation scheme Grade and the critical attribute Year, which is a non-key attribute that belongs to both relation schemes. As a consequence of the detected DB schema dependency, the following extended referential integrity constraints are generated:

- \( RID\_EXT\_Grade\_Student\_Course: \)
  \( \triangleright\triangleleft(Grade, Course)(((StudentId, Year)) \subseteq Student((StudentId, Year)); \) and
- \( RID\_EXT\_Student\_Grade\_Course: \)
  \( \triangleright\triangleleft(Student, Grade)(((CourseId, Year)) \subseteq Course((CourseId, Year)). \)

Here the sign \( \triangleright\triangleleft \) denotes the natural join. Also, the attribute Year is identified as a potential homonym, since it is a non-key attribute that belongs to different relation schemes. Only a designer can decide whether it is a real homonym, or not. If the attribute Year is not a homonym, then the DB schema remains A-dependent and the aforementioned extended referential integrity constraints preserve the data consistency for Year. However, if it is a homonym, a designer must introduce a new, renamed attribute that replaces the attribute Year either in the form type CEX, or the form type STU (Figure 8), and restart the DB schema design process.

Relation schemes Student and Major are also A-dependent owing to the existence of a critical attribute set \( \{FacId, DepId\} \). The critical attribute set belongs to both relation schemes, although it is not a key of either. The relation scheme Major\_Student references both Student and Major. The previously generated referential integrity constraint \( R_{Major\_Student: Major\_Student: Major\_Student: Major\_Student: Student}\) is transformed into the inclusion dependency:

- \( RID\_Major\_Student\_Student: \)
  \( Major\_Student((StudentId, FacId, DepId)) \subseteq Student((StudentId, FacId, DepId)); \)
and the following extended referential integrity constraints are generated:

- \( RID\_EXT\_Student\_Major\_Student\_Major: \)
  \( \triangleright\triangleleft(Major\_Student, Student)(((FacId, DepId, MajId)) \subseteq Major((FacId, DepId, MajId)); \) and
- \( RID\_EXT\_Major\_Student\_Major\_Student: \)
  \( \triangleright\triangleleft(Major\_Student, Major)(((FacId, DepId, MajId)) \subseteq Student((FacId, DepId, MajId)). \)
Figure 9. DB Schema Dependency and Homonym Diagnostic Report.

Figure 10 shows the closure graph of the potential DB schema generated by IIS*Case. Oriented edges of the graph represent interrelation constraints. IIS*Case displays basic referential integrity constraints, the referential integrity constraints based on non-trivial inclusion dependencies, and the inverse referential integrity constraints as edges of different colors. Basic referential integrity constraints are seen in Figure 10 as black edges, while the others are seen as greyed ones. Every edge of an inverse referential integrity (IRI\textunderscore Department\textunderscore Major or IRI\textunderscore Faculty\textunderscore Department) is followed by an edge that represents the appropriate basic referential integrity constraint (RI\textunderscore Major\textunderscore Department, i.e. RI\textunderscore Department\textunderscore Faculty). A referential integrity constraint based on
DEVELOPING COMPLEX DATABASE SCHEMAS USING FORM TYPES

Figure 10. The closure graph of the Faculty Organization DB schema.

non-trivial inclusion dependency can have identical referencing and referenced relation schemes, as it is the case with $RIJC\_Course\_Course$.

Step 2.3, Consolidation of the potential DB schema of the application system Faculty Organization and its subschema Student Service, is presented in the following section.

Detecting and resolving collisions

By selecting IIS*Case option DB Schema Analysis (Figure 6), we check the consistency of a DB schema and its subschemas. If collisions of a given type are detected, the appropriate report is generated.

Each collision report has the same structure in IIS*Case. The first part is a textual description of the report content. The second part is the content itself, i.e. a list of detected collisions. The third part is a textual description of the rule that must be satisfied to avoid collisions of a given type. The fourth part
is a list of examples that explain all common situations, which are recognized as collisions, and how they may be resolved.

The first step of a DB schema analysis is checking the consistency of attribute sets. In the case of consistency checking of the *Faculty Organization* DB schema and its sole subschema *Student Service*, the algorithm detects a collision, as indicated in Figure 6. The generated collision report is shown in Figure 11.

![Figure 11. The report on attribute set collisions.](image)

---

**Copyright © 2007 John Wiley & Sons, Ltd.**
A collision arose because the relation scheme Grade in the application subsystem Student Service contains the attribute CourseShortName, which does not belong to the corresponding relation scheme Grade in the application system Faculty Organization.

In order to resolve the collision, we suppose that a designer decided to return to step 1 of the DB schema design process (Figure 4), replace the attribute CourseShortName with CourseId in the form type STG (Figure 7), and pronounce it as a key of the component type GRADES. Also, he or she decided to introduce a new form type LIST OF COURSES, in order to enable handling pairs of values (CourseId, CourseShortName) in the subsystem Student Service. The new version of the application subsystem Student Service is shown in Figure 12. In this way, the second execution of the DB schema integration starts.

The next step is generating a new version of the DB schema (i.e. subschema) of the application subsystem Student Service, and then generating a new version of the potential DB schema of the application system Faculty Organization.

The second execution of step 2 (Figure 4) generates the following relation schemes of the subschema Student Service:

- Student\{StudentId, Year\}, \{StudentId\}, \{}\};
- Grade\{StudentId, CourseId, Grade, Date\}, \{StudentId+CourseId\}, \{}\}; and
- Course\{CourseId, CourseShortName, CourseName, Prerequisite\}, \{CourseId\}, \{}\};

and the set of interrelation constraints:
• \( \text{RI}_\text{Grade}\_\text{Student}: \text{Grade}[\text{StudentId}] \subseteq \text{Student}[\text{StudentId}]; \)
• \( \text{RI}_\text{Grade}\_\text{Course}: \text{Grade}[\text{CourseId}] \subseteq \text{Course}[\text{CourseId}]; \) and
• \( \text{RI}_\text{JC}\_\text{Course}\_\text{Course}: \text{Course}[\text{Prerequisite}] \subseteq \text{Course}[\text{CourseId}]; \)

while the potential DB schema of the application system \( \text{Faculty Organization} \) remains unchanged.

After generating relational DB schemas in the second execution of step 2, step 2.3, ‘consolidation’, i.e. DB schema analysis (Figure 6) is initiated again. This time, checking the consistency of the attribute sets succeeds, and analysis continues by checking the consistency of the sets of keys. However, a key constraint collision is detected and a collision report is generated (Figure 13).

The collision arose since the key \( \text{CourseShortName} \) in the corresponding relation scheme \( \text{Course} \) in \( \text{Faculty Organization} \) that was not declared as a key in the relation scheme \( \text{Course} \) in \( \text{Student Service} \). Suppose a designer decided to return to step 1 (Figure 4), and resolve the collision by declaring \( \text{CourseShortName} \) as a key of the component type \( \text{COURSE} \) in the form type \( \text{LOC} \) (Figure 12). A new version of \( \text{LOC} \) is shown in Figure 14.

Suppose that a designer of the application system \( \text{Faculty Organization} \) decided to simultaneously declare the attribute \( \text{Prerequisite} \) in the component type \( \text{COURSE} \) of the form type \( \text{CEX} \) (Figure 8) as a unique constraint. A new version of \( \text{CEX} \) is shown in Figure 15.

These changes in the conceptual model initiate the third execution of the DB schema integration process. Generating a new \( \text{Student Service} \) subschema introduces a sole change into the relation scheme \( \text{Course} \) by identifying \( \text{CourseShortName} \) as an equivalent key:

- \( \text{Course}([\text{CourseId}, \text{CourseShortName}, \text{CourseName}, \text{Prerequisite}],
\{\text{CourseId}, \text{CourseShortName}\}, \{\}\). \)

Generating a new potential DB schema of \( \text{Faculty Organization} \) also introduces a change into the relation scheme \( \text{Course} \) by identifying \( \text{Prerequisite} \) as a unique constraint:

- \( \text{Course}([\text{CourseId}, \text{CourseShortName}, \text{CourseName}, \text{Prerequisite}, \text{Lecturer},
\text{Year}, \text{Semester}], \{\text{CourseId}, \text{CourseShortName}\}, \{\text{Prerequisite}\}). \)

For the third time, DB schema analysis (Figure 6) is initiated from the beginning. Checking consistency of the attribute sets and consistency of the key constraints succeeds. This time, a collision of the unique constraint type is detected and a collision report is generated (Figure 16).

The collision arose since the unique constraint \( \text{Prerequisite} \) of the corresponding relation scheme \( \text{Course} \) in \( \text{Faculty Organization} \) was not declared as a unique constraint in the relation scheme \( \text{Course} \) in \( \text{Student Service} \). Suppose a designer decided to resolve the collision by declaring \( \text{Prerequisite} \) as a unique constraint in the component type \( \text{COURSE} \) of the form type \( \text{LOC} \) (Figure 14). A new version of \( \text{LOC} \) is shown in Figure 17.

Generating a new \( \text{Student Service} \) subschema in the fourth execution of the DB schema integration process introduces a sole change into the relation scheme \( \text{Course} \) by identifying \( \text{Prerequisite} \) as a unique constraint:

- \( \text{Course}([\text{CourseId}, \text{CourseShortName}, \text{CourseName}, \text{Prerequisite}],
\{\text{CourseId}, \text{CourseShortName}\}, \{\text{Prerequisite}\}); \)

while the potential DB schema of the application system \( \text{Faculty Organization} \) remains unchanged.

Copyright © 2007 John Wiley & Sons, Ltd.  
DOI: 10.1002/spe
DEVELOPING COMPLEX DATABASE SCHEMAS USING FORM TYPES

Figure 13. The report on key constraint collisions.
The DB schema analysis (Figure 6) provides automatic detection and resolution of null constraint collisions. Suppose that CourseName in the form type CEX of Faculty Organization (Figure 8) was declared as a not null attribute (\(\text{Null(CEX, CourseName)} = \bot\)), whereas the same attribute in the form type LOC of Student Service (Figure 17) can have null values (\(\text{Null(LOC, CourseName)} = \top\)).

IIS*Case detects a null constraint collision, automatically resolves it by converting CourseName in the relation scheme Course of Faculty Organization into an attribute with null values allowed, and generates a report with an ‘info’ message (Figures 18 and 19). This change makes using the form type LOC in Student Service possible for retrieval and update without having a real value for CourseName. On the other hand, when the form type CEX of Faculty Organization is used, a non-null value of...
### Developing Complex Database Schemas Using Form Types

**Figure 16. The report on unique constraint collisions.**

<table>
<thead>
<tr>
<th>Child application system</th>
<th>Relation scheme</th>
<th>Unique constraint</th>
<th>Corresponding relation scheme in Faculty Organization</th>
</tr>
</thead>
<tbody>
<tr>
<td>Student Service</td>
<td>Grade</td>
<td>[Date, StudentId]</td>
<td>Grade</td>
</tr>
</tbody>
</table>

**Rule**

Suppose there is a relation scheme $N_1$ in a child application system $S_1$, for which $N_2$ is the corresponding relation scheme in the parent application system $S$. A unique constraint $X$ from $N_2$ must be included in the relation scheme $N_1$ if:

- any attribute from $X$ also belongs to the attribute set of $N_1$ and an insert operation or modification of the attribute is allowed in $N_1$, and
- $X$ is not a key of $N_1$

**Example**

**Notation remarks:**
- $N(R, K, U)$ denotes a relation scheme with the attribute set $R$, the set of keys $K$ and the set of unique constraints $U$.
- $\text{Key}(N, X)$ denotes that $X$ is a key of $N$.
- $\text{Unique}(N, X)$ denotes that $X$ is a unique constraint in $N$.
- $\text{Role}(N)$ denotes a set of database operations that are allowed in a relation instance over $N$.
- $\text{Mod}(N)$ denotes a set of attributes from $R$ that are modifiable in a relation instance over $N$.

$S_1$, $S_2$ are the child application systems of an application system $S$.

$S_1$: $N_1((A, B, E), (A))$
- $\text{Role}(N_1) = \{r, i, m\}$ \quad // Allowed operations: Read, Insert and Modify
- $\text{Mod}(N_1) = \{B\}$ \quad // Modifiable attribute: $B$

$S_2$: $N_2((A, B, C, D, E), (A, CD), (BC))$

$S$: $N_3((A, B, C, D, E), (A, CD), (BC))$

**Collision:**

$N_3$ is the corresponding scheme for $N_1$ and $N_2$. Besides, $N_1$ does not contain the constraint $\text{Key}(N_1, BC)$. The constraint $\text{Unique}(N_1, BC)$ is not included in $N_1$ but it must be, because:

- the attribute $B$ is included both in $N_1$ and $N_3$,
- insert or modify of $B$ is allowed in $N_1$ and
- there is a constraint $\text{Unique}(N_3, BC)$.

Accordingly, the attribute $C$ must be included in $N_1$, too.
**Figure 17. A new version of the form type LOC in Student Service.**

*CourseName* must exist in each instance of CEX. This change, made by IIS*Case, does not affect the form types CEX and LOC in any way.

IIS*Case applies a special, non-null related check on those relation schemes in child applications systems that are declared for inserts. If anyone of them does not contain all non-null attributes of its corresponding relation scheme in the parent application system, a collision arises and the DB schema analysis stops. For example, suppose $\text{Null}(\text{CEX, Semester}) = \bot$ holds. If we had declared the COURSE component type of LOC in Figure 17 for inserts, then it would cause an error in the report shown in Figure 18.

After resolving null constraint collisions, the process of DB schema analysis goes on.

The next action in the same execution of step 2.3 (Figure 4) is checking the consistency of the sets of referential integrity constraints. This analysis may have one of the following three outcomes: no errors detected; warnings only detected; and errors detected. If the analysis detects no errors, or warnings only, IIS*Case considers it as successful unless a designer decides to resolve the warnings. If the analysis detects an error, a designer has to resolve collisions.

The operation **delete** is associated with the component type COURSE in the form type LOC in Student Service (Figure 17). Consequently, IIS*Case associates a delete operation with the Course relation scheme of the Student Service subschema. In the example considered, a warning is generated during the consistency checking of referential integrity constraints. The corresponding report is shown in Figures 20 and 21.

The warning arose owing to the referential integrity constraints, $\text{RI}_{\text{Exam,Course}}: \text{Exam}[\text{CourseId}] \subseteq \text{Course}[\text{CourseId}]$ and $\text{RI}_{\text{Timetable,Course}}: \text{Timetable}[\text{CourseId}] \subseteq \text{Course}[\text{CourseId}]$, in the application system Faculty Organization, which are not embedded in the subschema Student Service. They should be, because the relation scheme Course in Student Service may be used for deleting rows. The analysis also discovers that the relation schemes Exam and Timetable of Faculty Organization are relevant for that constraint but not included into Student Service. Since these are only warnings and not errors, a designer may decide to ignore them and finish the process of consolidation. If this is so, the last version of the Faculty Organization potential DB schema is pronounced as a final DB schema of the information system.
The situation when a subschema contains a referenced relation but not the referencing one, and deleting tuples of referenced relation is allowed in the subschema, represents the only situation when IIS*Case generates a collision warning and not an error. Strictly speaking this collision can be resolved either by (i) disallowing the delete operation, or by (ii) including the referencing relation scheme in the subschema. It may happen that the first option is not feasible, and that a repetitive application of the second option introduces a vast number of new relation schemes into the same subschema. To avoid a subschema overloading, we leave it to the designer to decide whether to resolve or to ignore a collision of that kind. Our justification to such an approach is the following:

- to initiate a delete operation, a user needs to see only the tuple to be deleted; and
- a referential integrity constraint is implemented anyway on a DB server, and it will prevent any attempt to delete a referenced tuple.
Figure 19. The report on null constraint collisions (Part 2).
Figure 20. The report on referential integrity constraint collisions (Part 1).
Figure 21. The report on referential integrity constraint collisions (Part 2).
The net result of ignoring a collision warning is that a user receives the information that their delete operation was unsuccessful after its submission. In this way, we trade off a small amount of user's comfort for a more reasonable software design.

In the case study, we have demonstrated only a selected number of possible situations that may arise during the process of a DB schema design in the IIS*Case environment. Generally, IIS*Case has a broader functionality and supports specifications of a number of concepts that are not discussed here. Our goal was to present only the main features of our approach.

**FINAL REMARKS**

The case study has introduced an example with only two application systems, with one being a sole child of the other. The same example could have been organized in a different way by defining three application systems: *Student Service*, *Faculty Organization*, and a new one with the name *Faculty Information System*. *Faculty Information System* would have an empty set of form types, and *Student Service* and *Faculty Organization* would be its child application systems. Despite the fact that the order of steps performed during DB schema integration would be slightly different, the same final DB schema would be obtained. In practice, there are often many ways of organizing an application system tree in order to obtain a desired result.

Although using our approach and IIS*Case the DB schema design proceeds more easily and quickly then otherwise, the design process may not be as simple as it seems at the first glance. For example, a designer may cause a new collision when attempting to resolve an existing one. That leads to at least one additional repetition of the DB schema integration process.

Furthermore, each interrupt of the integration process caused by a collision requires restarting the process from the point where the collision originated, which may be far away from the interrupt point. Each collision is resolved by making changes in the application system, in which a designer identified the origin of a collision. The integration process must be restarted from that application system and has to follow the same path as before until the point of the interrupt is reached again. It may happen that a collision that is detected at the very end of the integration process forces the redoing of almost the whole process again. That is a side-effect of our approach that designers have to take into account, if they want to perform the integration process in a formally correct manner. By doing so, they will have a guarantee that the integrated DB schema is formally correct. Consequently, the probability of achieving a high-quality DB schema is considerably higher than if they use an intuitive approach to DB schema integration, which would be based mostly on common sense.

Our approach and IIS*Case have been used in a large information system project for industry. The project had the goal to develop and implement an integrated information system of a railroad building and maintenance company. One of the authors of this paper led the project. Before the start of the project the company had only modest experience in using information technologies. Therefore, most of the application systems were developed from scratch. At the end of the design phase, the integrated DB schema contained 400 relation schemes. The implemented software of the information system included about 2000 screen form transaction programs, and 700 report programs. There were 20 application systems in the application system tree. At the end of the deployment, the system supported about 100 users. After seven years and a number of upgrades, the information system is still in use with its principal design unchanged.
The design and development team comprised 12 people, with roughly half of them being well experienced, and the other half being either modestly experienced or inexperienced. Eight of them were supposed to use our design approach and IIS*Case for the first time. They undertook five days of training for the approach and IIS*Case. After that, they were able to perform requirements engineering and DB schema generating. It took them approximately 10 man-days to design an application system. They found that the form type concept was easy to understand and convenient for communicating with the end-users. Generating a relational DB schema (Step 2.2, Figure 4) was an easy task, and the designers did not find any problems in reviewing the resulting DB schema, detecting semantic errors, and finding their causes in the form types. One of the authors performed the integration and consolidation (step 2, and particularly step 2.3 in Figure 4) through sessions with each of the designers. Resolving detected collisions was the hardest task at the start, because it was often hard to persuade a designer to make changes in the form types. However, as soon as the designers realized that the changes were justified, and comprehended how the collisions propagate from their source form types to relation schemes, they became able to anticipate the appearance of usual collision types in advance and to avoid them during the corrective form type design.

We gathered a similar experience using our design approach and IIS*Case in a number of other, smaller projects. Based on that, we are convinced that even fairly inexperienced designers supported by an expert and using the approach and IIS*Case are capable of producing an integrated DB schema of a high quality in a reasonably short time.

CONCLUSION

The form type concept is semantically rich enough to enable specifying an initial set of constraints such that it is possible to generate an implementation database schema automatically.

Design of external schemas, relying on high-level abstract data models, significantly facilitates identification of attribute and constraint sets. The approach presented here is based on the form type data model. From the designer’s point of view, the form type data model offers a simple way to define the initial set of attributes and constraints. The authors believe it to be an original approach, which cannot be found in the same form in similar tools. The approach does not require any advanced knowledge in the database area. It simplifies defining constraints, saves working time, and lowers the required level of designer’s intellectual effort, because it automates the majority of the tasks.

Detecting collisions of the most important constraint types is fully automated, which allows a designer to fully concentrate on collision resolution. Some of the collisions can even be resolved automatically, whereas most of the constraint collisions must be resolved at the conceptual level by a designer’s intervention. IIS*Case supports cooperative work of designers. In this way, IIS*Case facilitates attaining the most appropriate solutions.

Using IIS*Case, a designer may devote his or her time and intellectual power to analysis and modelling of business processes and rules. The database design of even complex information systems with database schemas composed of several hundreds of relation schemes with several thousand attributes is an easy task using our approach and IIS*Case, because the process of modelling is raised to the level that is closer to users without an advanced knowledge of database design. Therefore, we believe that our approach and IIS*Case are convenient for application in agile software development.
FURTHER RESEARCH AND DEVELOPMENT

IIS*Case is aimed at supporting the development of the complete application software for an information system. At present, IIS*Case R.6.2 produces a formal specification of an implementation database schema. It also has an SQL generator that supports the generation of SQL specifications of a database schema for different DBMSs. Further research and development efforts are oriented towards extending current functionality. Accordingly, we are planning or already working on:

- visualization of form types;
- development of a tool for creating common models of user interface (UI) design;
- implementation of an application generator; and
- further improvement in integration and consolidation algorithms.

One of the goals is to provide visual design and specification of form types by using a graphical editor, flexible enough to support modelling of varying user forms.

We believe that the software applications of an information system should conform to a common standardized model of a UI. Therefore, we are working on a tool for creating common models of UI design. This tool is aimed at supporting the development of various templates of UI design. The templates may be further applied in the process of application generation in order to produce software applications that conform to the same UI model.

In addition to constraints that are the basis for a database schema design, form types carry additional information about transaction programs and their screen forms. This enables the implementation of a code generator within IIS*Case, which will be able to generate transaction programs. These processes are already specified in [5], in which the problem of formalizing and generating programming specifications and applications of an information system, using XML technology, are discussed. These processes should be improved and implemented in a new release of IIS*Case.

At present, IIS*Case does not detect collisions of check constraints, extended referential integrity constraints, and inverse referential integrity constraints. Further work will extend the consolidation algorithms to support these constraint types.

ACKNOWLEDGEMENT

The authors would like to thank Raymond Nickson from the University of Wellington for his valuable proof reading.

REFERENCES