

Chapter XIX

A Concept-Based Query Language Not Using Proper Association Names

Vladimir Ovchinnikov, Lipetsk State Technical University, Russia

ABSTRACT

This chapter is focused on a concept-based query language that permits querying by means of application domain concepts only. The query language has features making it simple and transparent for end-users: a query signature represents an unordered set of application domain concepts; each query operation is completely defined by its result signature and nested operation's signatures; join predicates are not to be specified in an explicit form, and the like. In addition, the chapter introduces constructions of closures and contexts as applied to the language which permits querying some indirectly associated concepts as if they are associated directly and adopting queries to users' needs without rewriting. All the properties make query creation and reading simpler in comparison with other known query languages. The author believes that the proposed language opens new ways of solving tasks of semantic human-computer interaction and semantic data integration.

INTRODUCTION

Conceptual models serve for application domain modeling as opposed to means of system implementation modeling. A conceptual model does not concern implementation details and describes an application domain's essence. Conceptual models underlie conceptual query languages that are meant for querying schemas of the models (here and throughout the chapter, a model is considered to be a mean of modeling, and a schema

is considered to be a result of modeling). The languages have dual use. On the one hand, conceptual queries play the key role in constraint formalization—any constraint can be formulated as a query and an assertion upon it. On the other hand, the queries can be used for requesting data from an information system wrapped by a conceptual schema. In both cases, conceptual query transparency and simplicity are very important.

Aiming at more transparency and simplicity of conceptual queries, the author proposes Semantically Complete Query Language (SCQL) (Ovchinnikov, 2004b, 2005b; Ovchinnikov & Vahromeev, 2005). The language is founded on the semantically complete model (SCM) (Ovchinnikov, 2004a, 2005b, 2004c), the main property of which is semantic completeness that endows the model and query language with their names. A schema of the model is a set of application domain concepts, concept associations, and constraints defined over. The semantic completeness property implies a SCM schema does not include associations describing interrelation of application domain concepts differently; in other words, each association describes semantics of concept interrelation completely (more precise definition will be given in the section, restrictions imposed on underlying model). The main consequence of the property is that associations are based on unique (within a schema) concept sets; an association is identified with a set of underlying concepts, and not a proper name. The consequence is the creation of SCQL that uses concept sets for referring to associations. The language permits querying by dint of application domain concepts completely; proper names of associations are not used within it.

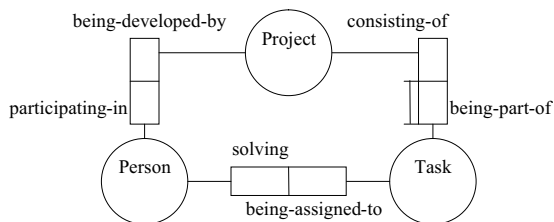
There are several other properties of SCQL that resulted in more simplicity and transparency of its queries: each query operation is completely defined by its signature¹ and nested operations' signatures; a signature of any query is an unordered set of application domain concepts; and join predicates have not to be specified in an explicit form. In addition, this chapter introduces conceptions of closures and contexts as applied to the language. The conceptions permit querying some indirectly associated concepts as if they are associated directly and adopting queries to users' needs without rewriting. All the properties make query creation and reading simpler in comparison with other known query languages, which will be proved in the subsequent sections. The author believes that all these properties and others discussed permit usage of the language by end-users who are not specialists in information technologies (IT).

The chapter considers restrictions imposed on an underlying model by SCQL, the way of referring to associations within it, the structure of SCQL expressions, and the context mechanism. All ideas are illustrated using the running example introduced in the next section. Finally, the chapter shows the ways of application and development of the query language.

QUERY SIMPLIFICATION METHODS

Now there exist many conceptual and data models and modeling approaches: entity-relationship (ER) (Chen, 1976; Chen, 1981), object-role modeling (ORM) (Bronts, Brouwer, Martens, & Proper, 1995; Halpin, 1995, 2001) and its particular cases (Brouwer, Martens, Bronts, & Proper, 1994; Bommel, Hofstede, & Weide, 1991; Halpin & Orłowska, 1992; Hofstede & Weide, 1993; Nijssen & Halpin, 1989; Troyer, 1991), fully communication oriented information modeling (FCO-IM) (Bakema, Zwart, & Lek, 1994), conceptual

Figure 1. ORM model of project management domain



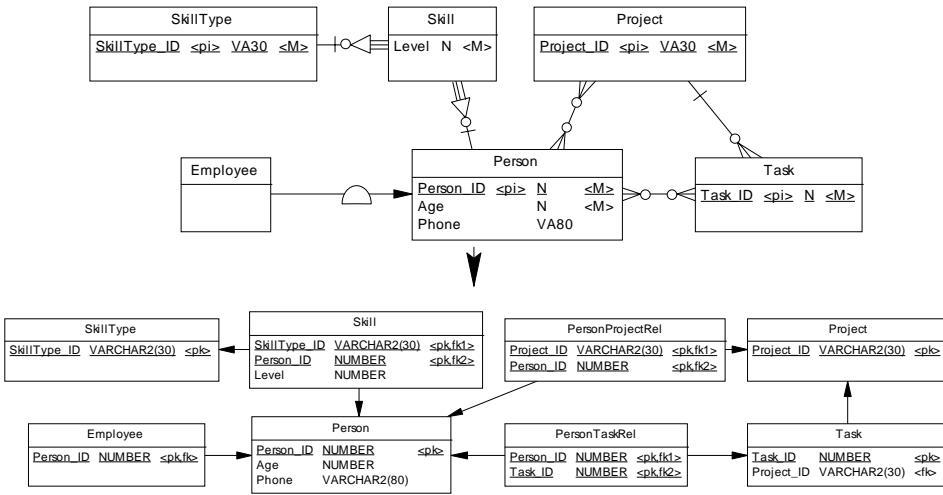
graphs (CG) (Dibie-Barthelemy, Haemmerle, & Loiseau, 2001), Web Ontology Language OWL (W3C, 2004b), resource description framework (RDF) (W3C, 2004a), relational model (RM) (Codd, 1979), and others. Almost every existing model underlies one or several query languages used for accessing information through the models, for instance, ORM underlies LISA D (Hofstede, Proper, & Weide, 1993, 1996) and Conquer-II (Bloesch and Halpin, 1997), RDF underlies RDQL (Seaborne, 2004), RDL and other query languages, RM underlies relational algebra (Codd, 1972) and partially SQL, and so forth. Unfortunately, all existing query languages are not sufficiently transparent for end-users not being specialists in IT, though several simplification methods were applied to some of them.

Query simplification can be achieved by using natural names for entities and relations when modeling and querying (Halpin, 2004; Hofstede, Proper, & Weide, 1997; Owei, 2000; Owei & Navathe, 2001b). The method significantly simplifies end-user work as interaction with a system takes place directly in application domain terms. Examples of such query languages are LISA D (Hofstede, Proper, & Weide, 1993, 1996) and CQL (Owei & Navathe, 2001b). But this simplification is not structural: the query structure remains complex. Let us illustrate the fact with the example of project management domain. Persons, tasks, and projects are the main entities of the domain: projects consist of tasks that are assigned to persons; persons can participate in project teams directly. The application domain has the formalization in ORM as seen in Figure 1.

The query "select all tasks assigned to persons participating in the project MES's team" is formulated in LISA-D as "Task being-assigned-to Person participating-in Project MES." The path expression has the following complexity factors: a) the order of entities and roles is important and should be kept correct; b) the appropriate role names should be remembered precisely (for instance, "being-assigned-to" or "participating-in"). A user is not insured against creation of senseless queries like "Task solving Person," "Person consisting-of Project," or mistakes like "Person solved Task," as a result of incorrect usage or remembering of the role names. Using the SCQL language discussed further, the query is formulated as (Task–Person–Project="MES"). Here proper relation or role precise names are not used, and one does not have to remember them.

Unfortunately, LISA-D did not become an industrial standard for information system development and user interaction, and it is slightly supported by tools. Now the

Figure 2. ER and relational schemas of project management domain



standard is SQL. Therefore, let us use SQL for the comparison purpose below. It is allowable because SCQL and SQL applications have one common field—both languages can be used as a mean of end-user interaction with an information system. In the case of SCQL, such information systems are to be wrapped by SCM and may be backed by relational or other DBMS [prototype system implementation can be found in Ovchinnikov (2005a)].

The project management domain can be formalized, using ER (Chen, 1976, 1981) and relational model, as shown in Figure 2. To use the example as a running one, we have introduced new entities: person’s phone, age, skills, and an employee as a particular case of a person.

The query “select all tasks assigned to persons participating in the project MES’s team” considered above is formulated in SQL as follows:

```
SELECT Task_ID FROM PersonTaskRel ptr, PersonProjectRel ppr
WHERE ptr.Person_ID = ppr.Person_ID AND ppr.Project_ID = 'MES'.
```

The given SQL query has the following complexity factors in comparison with the SCQL query (Task–Person–Project=“MES”):

- the join predicate “ptr.Person_ID = ppr.Person_ID” is defined explicitly;
- the appropriate precise table names should be remembered;
- the query’s signature is lacking in semantics since it consists of abstract columns not associated with the application domain’s concepts;
- names of fields and tables are noticeably far from the natural language.

Finally, the SCQL query is shorter and easier to understand.

Known query languages use proper names for referring to associations (relations, fact types); one has to remember many precise names to formulate queries using the languages. The reason lies in models underlying the languages: ORM, ER, RM, and others. The models require identification of relations by their proper names. Any two entities of a schema of the models can be associated in many ways, and each of the ways takes its own unique name. As a result, one cannot think about entities as if they are merely associated. At the same time, it is not necessary to remember a precise name of the association of “Person” and “Task” if one refers to it by (Person, Task) as the proposed language implies. Such language behavior impacts properties of the underlying model, which will be considered in the next section.

Not all associations can be named clearly and shortly. Sometimes full names of associations are whole sentences actually enumerating participated concepts and only. For instance, the association of “Person” and “Task” can be named “Persons solving tasks,” “Tasks being solved by persons,” or “Assignments of tasks to persons.” Formulating a query within any known query language, one should remember the way of association naming. This is not necessary when a concept enumeration is used for referring to an association, for instance, (Person, Task). Detailed discussion of referring to associations within SCQL will be given in the section, “Association Referring And Context Mechanism Within SCQL Expressions.”

Many query languages have another complexity factor—query signatures are not based on application domain concepts; in such languages, interpretation of a query result is completely determined by a structure of the query. For instance, the column “Task_ID” in the previous SQL query can mean anything, even phone number. One should analyze the query’s structure to understand the real meaning of the column. Moreover, one is not insured against formulation of senseless queries, for instance, joining the tables “Skill” and “Person” with “Age = Level.” As a result, the languages are too complicated for end-users. The offered solutions for the problems will be discussed in the following sections.

Another way of query simplification is usage of a GUI application concealing query complexity, as, for instance, Conquer-II (Bloesch & Halpin, 1997) and OSM-QL (Embley, Wu, Pinkston, & Czejdo, 1996) offer. Using intuitively clear interface elements like trees, one can easily construct conceptual queries. Nevertheless, the approach’s extent of simplification has the limit imposed by strong impact of a query language structure to GUI: tree node types, node connectivity, and node attributes are dictated by the structure. Since each operation of the language proposed is completely defined by its resulting signature and nested operations’ signatures, the author believes the language has simpler structure than existing query languages, well suits the purpose of GUI-based query languages, and should be developed in that direction in the future.

Existing query languages still remain complex for end-users as they have the following main complexity factors: a) queries are formulated using association proper names, and not application domain concepts; b) queries have structure including many complicated elements; c) there is no context mechanism that would permit using some indirectly associated concepts as if they are associated directly according to pre-adjusted context. The chapter introduces Semantically Complete Query Language (SCQL) which attempts to solve the complexity factors.

Let us summarize characteristics of SCQL and the well-known query languages LISA-D, Conquer, and SQL (see Table 1). The languages were selected as they are

Table 1. Summary of characteristics of the languages LISA-D, Conquer, SQL, and SCQL

Characteristic	LISA-D	Conquer	SQL	SCQL
Declarative queries	+	+	+	+
Natural names for entities and associations (relations)	+	+	-	+
GUI-based query formulation	-	+	-/+	-
Semantic result signatures (referring to domain concepts)	+	+	-	+
Purely concept-based query formulation (uselessness of proper association names)	-	-	-	+
Capability of implicit join predicates	-/+	-/+	-	+
Prohibition of senseless queries	-/+	-/+	-	+
Formulation of queries as concept chains	-/+	-/+	-	+
Formulation of join-like queries as a resulting signature merely	-	-	-	+
Query adaptation without rewriting	-	-	-	+

representative specimens of the very different query language categories. Analyzing the table, one could conclude that the most important distinction of SCQL is pure concept-based query formulation without resorting to association proper names. In the next sections, all the listed characteristics will be considered in detail, in addition to GUI-based query formulation, which is perspective of SCQL development.

RESTRICTIONS IMPOSED ON UNDERLYING MODEL

Disusing of proper association (relation, fact type) names promises the most noticeable increase of query language simplicity. The only way of referring to associations without the use of explicit names is to use concept (entity, object type) combinations as references to associations so that each concept combination gets identification of an appropriate association. The identification would imply a sequence of concepts, but this method is not transparent. Therefore, the language being proposed uses the

method of relation identification by means of sets of application domain concepts and does not use proper association names.

As a result, not any model can be used as basis for the query language; such model has to permit identification of relations by domain concept sets. A model having the identification property was proposed by Ovchinnikov (2004a, 2005a) and was named the semantically complete model (SCM). Moreover, SCM is more restricted than the identification property imposes—it is semantically complete. The semantic completeness property means that a) within a schema, each association is uniquely identified with a set of concepts underlying it, b) an association can not be based on a concept set being a proper subset of another concept set underlying another association of the same schema; in other words, each association describes semantics of concept interrelation completely. Any schema that satisfies the semantic completeness property (SCM schema) also satisfies the identification property since each association covers a unique set of concepts.

SCM is a full-scale modeling technique textual notation that is near to natural languages [see Ovchinnikov (2004a, 2004b, 2005a) for details]. Continuing the above-running example, let us present a SCM schema of the example domain in the textual notation:

Person solves Tasks [Task]

Person has a Phone →

Person has a Skill Level for a Skill Type

Employee is a Person ≡

[(Person, Skill Type) → Skill Level]

Project consists of Tasks ← [Task]

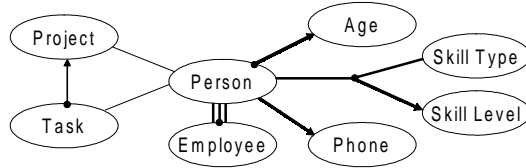
Person is of Age →

Project has a team of Persons [Team]

Here the associations and concepts are self-described as each association is represented by a sentence where application domain concepts are marked with capital first letters. The most general constraints are given within the sentences: functional constraints of binary associations (→), equivalent constraints of binary associations (=), mandatory constraints (_). For instance, the association “Person is of Age →” is constrained as “each person must correspond to only one age,” the association “Employee is a Person =” is constrained as “each employee must correspond to only one person and a person can correspond to only one employee,” and the association “Person solves Tasks” is not constrained at all.

More complex constraints are placed in square brackets after sentences with intend, for instance, “each combination of person and skill type can determine only one skill level” is formulated as “[(Person, Skill Type) → Skill Level].” Such constraints can be a lot more complex when being based on SCQL queries or statements formulated using SCQL-extended predicate calculus. The running example does not include all existing types of SCM constraints. Detailed definition of textual and graphical SCM notations, including the constraint language, is out of the scope of the chapter. The same SCM schema in graphical notation is presented in Figure 3.

Figure 3. Graphical notation of SCM schema of project management application domain



One can see from Figure 3 that SCM graphical notation is an extension of a type of hypergraph notation: concepts are nodes and associations are edges. Concepts are designated with ellipses and associations – with lines or star-lines connecting appropriate concepts. General constraints are placed upon concepts and associations: functional constraints as arrows, equivalent constraints as triple lines, mandatory constraints as dots. For instance, the association “Person has a Skill Level for a Skill Type” is designated with a star-line pointed at “Skill Level” as it is constrained with “[(Person, Skill Type) → Skill Level].”

Any SCM schema is a set of associations based on sets of concepts; also a SCM schema includes a set of constraints, but this question is out of the scope of the chapter. Let m be a set of SCM schemas, a be a set of associations, c be a set of concepts. Then $ma \subseteq m \times a$ determines correspondence of associations and schemas, and $ac \subseteq a \times c$ determines correspondence of concepts and associations. The association identification constraint “a model cannot have two associations based on the same set of concepts” can be formulated as follows:

$$[C1] \quad m \quad m \quad a \quad a \quad a \quad a \quad \begin{matrix} m,a & ma & m,a & ma \\ c | a,c & ac & c | a,c & ac \end{matrix}$$

The main property of SCM is semantic completeness, which means that within a schema there is no an association based on a concept set being a proper subset of a concept set of another association. This restriction guarantees that each association defines semantics of interrelation of underlying concepts completely.

$$[C2] \quad m \quad m \quad a \quad a \quad a \quad a \quad \begin{matrix} m,a & ma & m,a & ma \\ c | a,c & ac & c | a,c & ac \end{matrix}$$

The identification constraint C1 is sufficient for referring to associations without using proper names and, therefore, it is sufficient for creation of a query language not using proper association names. The semantic completeness constraint C2 is introduced since it increases schema and query simplicity and transparency; one can think about

interrelation of a set of concepts as complete phenomenon, knowing that there are no alternatives for this interrelation (Ovchinnikov, 2004b). This constraint impacts the context mechanism, which will be discussed in the next section.

The conceptual query language based on SCM and not using proper association names for query formulation was named the Semantically Complete Query Language (SCQL) (Ovchinnikov, 2004b, 2005a) and will be discussed in the following sections.

ASSOCIATION REFERRING AND CONTEXT MECHANISM WITHIN SCQL EXPRESSIONS

As a result of the identification constraint C1, SCQL uses concept sets for referring to associations and not proper names. SCQL provides for simple notation of such references, namely, enumeration of concepts by comma in round brackets; concept order in the enumerations is not important. For instance, both of the references (Person, Skill Level, Skill Type) and (Person, Skill Type, Skill Level) are correct and point to the same association. One can see that one does not have to remember proper association names to refer to the association and must only know the fact of interrelation of the concepts.

A reference to an association is considered as a selection of all its instances. For example, the expression (Person, Skill Level, Skill Type) is a selection of all instances of the appropriate association. The analogous SQL query is the following: “SELECT SkillType_ID, Person_ID, Level FROM Skill.” One can see from the example that the SQL expression includes the proper name of the table “Skill,” while the appropriate SCQL expression has no such element.

Here, a composition operation of SCQL can be considered as a mathematical composition (a natural join) of subqueries (see the next section for details). When a composition operation is built over selections of associations, it uses direct references to associations. For this case, there are two special notations that make an expression more simple and transparent for end-users: path and star notations. Each notation has its own scope of application where it is the most usable.

The path notation is used when several binary associations forming a connected chain are composed. A path expression is a chain of concepts separated by dashes. Each adjacent concept pair is considered as a concept set referring to an appropriate association. Therefore, a chain as a whole is a composition of all associations referred by adjacent pairs. For instance, the expression (Person–Task–Project) selects project and persons for each task by composing the associations (Person, Task) and (Task, Project). One can see that SCQL chains have no any attributes besides concepts themselves as opposed to, for example, LISA-D where names of used relations (predicators, saying more precisely) are to be indicated explicitly: “Person solving Task being-part-of Project.” SCQL path expressions can be written down starting from any edge concept, for example, (Project–Task–Person) is equivalent to (Person–Task–Project). The analogous SQL query is as follows:

```
SELECT t.Project_ID, t.Task_ID, ptr.Person_ID
FROM Task t, PersonTaskRel ptr WHERE t.Task_ID = ptr.Task_ID.
```

This SQL expression has the following complication factors that the above SCQL

expressions do not have: a) the resulting signature is not semantic since a result column could mean anything (for example, `Person_ID` could mean even “phone number”); one must analyze the SQL expression structure to understand the real semantics of each column; b) the explicit join predicate “`t.Task_ID = ptr.Task_ID`” has been defined; and c) the proper table names “Task” and “PersonTaskRel” have been used.

The star notation is used when several binary associations forming a star with one central concept are composed. A star expression is a comma-separated list of non-central concepts in square brackets chained with a central concept (by means of dash). For instance, one can use the star expression (Person-[Project, Phone]) instead of the path expression (Phone-Person-Project). The star notation is the most convenient when there are more than two non-central concepts in a star. The star notation has the same advantages relative to analogous SQL and LISA-D queries as the path notation.

One concept can play several roles within an expression, for example, when using one association several times. For this purpose, SCQL introduces the concept of a “role concept.” A role concept is a concept extended with a role name that indicates the concept’s role in a given expression. Role names are placed in round brackets after concepts if different roles are necessary. For instance, consider the expression (Project(Task’s)-Task-Person-Project(Person’s)). Here both projects are semantically diverse columns of the expression and have the role names “Task’s” and “Person’s.” The analogous SQL query is as follows:

```
SELECT t.Project_ID, t.Task_ID, ppr.Person_ID, ppr.Project_ID
FROM Task t, PersonTaskRel ptr, PersonProjectRel ppr
WHERE t.Task_ID = ptr.Task_ID AND ptr.Person_ID = ppr.Person_ID.
```

If one does not use the roles in the expression, it becomes cyclic with one project column: (Project-Task-Person-Project), which reads as “select persons with their tasks being part of projects of which the persons are members.” Since the expression is cyclic, it can be equivalently reformulated starting from any concept, for instance, as (Person-Task-Project-Person). In both cases of cyclic expressions, their resulting signatures contain only these three elements: “Person,” “Task,” and “Project.” The analogous SQL query is the following:

```
SELECT t.Project_ID, t.Task_ID, ppr.Person_ID
FROM Task t, PersonTaskRel ptr, PersonProjectRel ppr
WHERE t.Task_ID = ptr.Task_ID AND ptr.Person_ID = ppr.Person_ID
AND t.Project_ID = ppr.Project_ID
```

As the SQL query is cyclic, it contains the additional condition “`t.Project_ID = ppr.Project_ID`” that completes the cycle, and it has the only resulting “Project_ID” column.

Saying this formally, let rc be a set of role concepts, rn be a set of role names. Then the maps $rcc : rc \rightarrow c$ and $rcrn : rc \rightarrow rn$ reflect the facts that a role concept pertains to a concept and can have a role name; at that, each role concept is to pertain to a concept:

$$[C3] \quad \forall rc' \in rc \exists c' \in c \{ (rc', c') \in rcc \}$$

Concept enumerations are used within SCQL not only for referring to associations, but also for requesting interrelation of indirectly associated concepts by using the context mechanism of SCQL. The mechanism increases simplicity and transparency of queries to a greater extent since it permits omitting “trivial” inter-concept transition details. For example, if (Employee, Person) and (Person, Phone) are included to the current context, one can execute the query “select phones of employees” using (Employee, Phone) or (Employee–Phone) instead of (Employee–Person–Phone). Here the transition “Employee–Person–Phone” is considered as “trivial” and therefore can be shortened to “Employee–Phone.” Comparing the query (Employee–Phone) and the analogous SQL query: `SELECT e.Person_ID, p.Phone FROM Employee e, Person p WHERE e.Person_ID = p.Person_ID`, one can conclude that the SCQL query is a lot more simple and transparent than the SQL query. The analogous LISA-D query is also more complicated than the SCQL query: “Employee being Person having Phone.”

The context mechanism permits for some composition-projection queries to be shortened to simple enumeration of required concepts. The core concepts of the context mechanism are “association closure” and “execution context.” An association closure serves as an agreement on query shortenings and is characterized by unity of effect, that is, either all or none of shortenings implied by the agreement take effect. An association closure is defined over a SCM schema and is a set of associations of the schema. An execution context is a set of association closures or associations directly. An SCQL query-execution system has a single execution context at a time named as current one. The current context is used for executing any shortened SCQL query.

The context mechanism increases query transparency and simplicity; a composition-projection query can be shortened to simple concept enumeration. Therefore, a concept enumeration can mean a selection of an association as well as a shortened query. If the queried schema has an association based on the specified concept set, then the enumeration is considered as an association selection; otherwise, the enumeration is considered as a shortened query. For example, the query (Employee, Phone) is a shortening of the composition (Employee–Person–Phone), while (Person, Phone) is the reference to the appropriate association.

Any shortened query is subject to execution in the following way. Consider as a hypergraph all associations included to a context directly or indirectly by dint of closures. Pick out all connected sub-hypergraphs existing in the hypergraph. Each of the connected hypergraphs has its own set of concepts underlying its associations. If the desired shortened query enumerates concepts pertaining to different connected hypergraphs, it is concluded that the query is mistaken and cannot be executed. Otherwise, it is taken as a minimum set of associations that connect the required concepts, including all alternative connecting paths. The taken associations are composed and then projected by the required concepts. The result of the projection is the result of the shortened query.

Using the mechanism, all non-cycle path and star expressions can be written as simple enumerations of required concepts. For instance, the shortened query (Employee, Phone) is executed as a composition of the associations (Employee, Person) and (Person, Phone), and then the result is projected on “Employee” and “Phone.” The context mechanism serves for similar purpose as the abbreviated concept-based query language presented in Owei and Navathe (2001a) and Owei, Navathe, and Rhee (2002) that does

not require entire query paths to be specified but only their terminal points. Formally, let ac be a set of closures and cx be a set of contexts. Then $aca \subseteq ac \times a$ defines associations included to each closure and $cxac \subseteq cx \times ac$ defines closures constituting each context.

Association closures can be of different types. There can be some default closures for each SCM schema; the default closures are always included to the current context. Other closures are to be uniquely named since they are to be included to or excluded from a context explicitly. Both the default closure set dac and the named closure set ncc are subsets of the general closure set: $dac \subseteq ac$, $nac \subseteq ac$. A set of associations included with a closure can be specified explicitly or can be calculated from a schema according to an algorithm. For instance, a closure can be calculated as all associations not part of cycles on a schema. The calculated closure set cac is also subset of the general closure set: $cac \subseteq ac$.

Closures are designated on a SCM schema as follows. Associations of the main default closure are designated with bold as it is done in the running example above. Associations of named closures are followed by closure names in square brackets, as, for instance, the association “Project has a team of Persons [Team].” If an association is included with more than one named closure, they are enumerated with comma within the brackets.

The simplest context is an empty one when closures are not used and all concept enumerations refer to associations directly, as, for instance, (Person, Phone). Closures are added to and removed from the current context explicitly. Even if the context is not empty, one may decide not to use the context mechanism by writing full queries without shortenings. This is recommended if a query is not to change semantics when changing the current context; otherwise, if a query must be context-sensitive, it should be written down using context-sensitive shortenings.

The context mechanism makes SCQL more flexible, but if one uses the context mechanism heedlessly, query semantics can change unpredictably. Therefore, context change is to be closely controlled. For instance, if the current context contains the closure “Team” (in addition to the default closure of course) of the running example, the query (Project, Phone) or (Project–Phone) will select all phones of persons being members of project teams. If the current context contains the closure “Task,” then the same query (Project–Phone) will select all phones of persons solving tasks of the projects. So the query has different semantics in different contexts. One could make semantics stable if he/she would write it fully as the query (Project–Person–Phone) for the first semantics or as the query (Project–Task–Person–Phone) for the second semantics. Translating the shortened query (Project–Phone) to SQL, one gets two different SQL queries depending on the current context. The first SQL query is:

```
SELECT e.Person_ID, ppr.Project_ID FROM Employee e, PersonProjectRel ppr
WHERE e.Person_ID = ppr.Person_ID,
```

and the second one is:

```
SELECT e.Person_ID, t.Project_ID FROM Employee e, Task t, PersonTaskRel ptr
WHERE e.Person_ID = ptr.Person_ID AND ptr.Task_ID = t.Task_ID.
```

Both queries are a lot more complicated than the SCQL query (Person–Phone).

Context mechanism is very useful when a schema is evolving. If one modifies a schema not removing concepts and not changing their semantics, the modification can be done absolutely transparently by dint of default closure configuring. For instance, introduce a new concept to the running example: “Communication Address,” and replace the association “Person has a Phone \rightarrow ” of the schema with the following associations:

Person has a Communication Addresses –
Phone is a Communication Address °
 [(Person–Phone): Person \rightarrow Phone]

Since the new associations are in the default closure (they are in boldface)—and so always in the current context—all queries that used the association (Person, Phone) do not change their semantics. The concept enumeration (Person, Phone), which was the association selection, becomes a shortening for the composition (Person–Communication Address–Phone) and subsequent projection on the concepts “Person” and “Phone.” Therefore, the modification has passed unnoticed by schema users.

SCQL context can be created according to different strategies. An obvious strategy is to reflect users’ preferences for data browsing. This approach is suitable for simple queries when users go from one concept to another without writing complex expressions. In this case, a context can be changed explicitly or automatically by using browsing statistics, for instance, an association usage frequency.

Another strategy of context creation aims to reflect shortenings generally accepted by a community or an application domain. The generally accepted shortenings underlie default closures; other shortenings underlie several named closures and are optional for some part of a community or an application domain. The options are activated when necessary by user or automatically.

And the last strategy can be used in natural language recognition systems. The strategy implies a context changes dynamically for each new text part. According to this strategy, a context of a previous text part is used as basis for a context of a next text part and the last context is modified by using some statistics of both text parts. Context mechanism of SCQL is unique; other known query languages have no such mechanism at so deep an architectural level; context changes do not require query rewriting.

SCQL EXPRESSION STRUCTURE AND PROPERTIES

This chapter is focused on the following main SCQL property: queries are formulated by using application domain concepts completely; the property is guaranteed by the fact that associations are identified by concept sets; and it increases transparency and simplicity of query expressions, especially when using contexts, path and star expressions. In addition, SCQL has other interesting properties based on characteristics of its operations that are considered below.

An expression of any query language represents a tree of operations. A set of possible operation types varies from one language to another, but leaf operations always are selections from relations of an underlying schema. Let e be a set of expressions, o

be a set of operations, and ot be a set of operation types. Then $oe: o \rightarrow e$ determine operations of each expression, and $oot: o \rightarrow ot$ determine an operation type for each operation. Leaf operation types are subset of all operation types ($lot \subset ot$) and leaf operations are subset of all operations ($lo \subset o$). It is true that all and only leaf operations are to be of leaf operation types:

$$[C4] \quad \forall o' \in o \forall ot' \in ot \left\{ \begin{array}{l} (o', ot') \in oot \rightarrow \\ \{ \{ (o' \in lo \wedge ot' \in lot) \vee (o' \notin lo \wedge ot' \notin lot) \} \} \end{array} \right\}$$

Operations can be nested to other operations: $oo: o \rightarrow o$. All and only leaf operations have no nested operations:

$$[C5] \quad \forall o' \in o \left\{ \begin{array}{l} (o' \in lo \rightarrow \{ o'' \mid (o'', o') \in oo \} = \emptyset) \wedge \\ (o' \notin lo \rightarrow \{ o'' \mid (o'', o') \in oo \} \neq \emptyset) \end{array} \right\}$$

A signature $sign$ of any SCQL operation is a set of role concepts: $osign: o \rightarrow sign$, $sign \subseteq rc$. Each SCQL operation is to have a signature:

$$[C6] \quad \forall o' \in o \exists sign' \in sign \{ (o', sign') \in osign \}$$

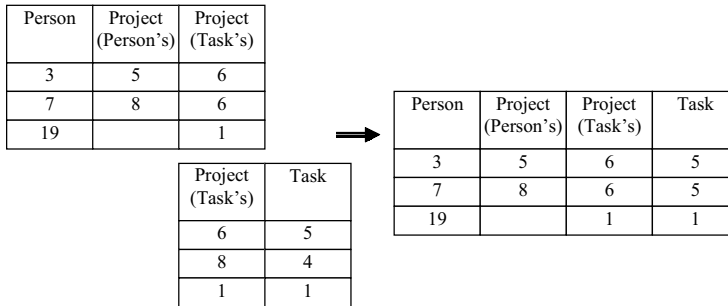
SCQL provides for the following operation types serving as non-leaf ones: composition, transformation, union, and minus. Operations of the types will further be named as composition operation, transformation operation, and so on. Let $comp$ be a set of composition operations, $trans$ be a transformation operation set, $union$ be a union operation set, and $minus$ be a minus operation set. All they are subsets of the general operation set: $trans \subset o$, $union \subset o$, $minus \subset o$, $comp \subset o$; and they are non-leaf operations:

$$[C7] \quad \forall o' \in (comp \cup trans \cup minus \cup union) \{ o' \notin lo \}$$

A composition operation is a mathematical superposition defined over role concepts as sets and SCQL subqueries as relations. A composition operation fulfills join-like transformation of nested operations: a) selects all instances, having the same values of identical role concepts, from Cartesian product of nested operations; and b) projects the result to avoid duplication of role concepts. The composition is analogous to the natural join of the relational algebra (Codd, 1972), but there is the following important distinction: composition fulfillment considers coincidence of application domain concept identities, while natural join fulfillment considers coincidence of attribute names. The natural join is not semantic as attribute names within Relational Model are not associated with application domain concepts directly. Two attributes representing one concept can have different names, and two attributes having the same name can represent different application domain concepts. At the same time, composition operations are semantic as they are based on application domain concepts directly.

For example, if given two nested operations with signatures (Person, Project(Person's), Project(Task's)) and (Project(Task's), Task), then their composition

Figure 4. SCQL composition fulfillment example



has the resulting signature (Person, Project(Person's), Project(Task's), Task), as shown at the Figure 4.

The composition has three different notations, two of which, path and star notations, were discussed above; those two notations are applicable to binary associations only. There is another notation that is applicable to any subqueries and, therefore, is the most general one. Using the general notation, one composes several subqueries by enumerating them in round brackets with comma. For instance, the query “((Employee, Person), (Person, Skill Level, Skill Type), (Person, Phone))” is composition of three association selections: “(Employee, Person),” “(Person, Skill Level, Skill Type),” and “(Person, Phone).” A composition operation does not have any parameters besides a set of nested operations. Just this fact enables all the notations: general, path, and star ones.

Composition signatures contain a union of role concepts of nested operation signatures and do not include one role concept several times:

$$\forall comp' \in comp \forall sign' \in sign$$

$$[C8] \left\{ \begin{array}{l} (comp', sign') \in osign \rightarrow \\ sign' = \bigcup \{ sign'' \mid \exists o' \in o \{ (o', comp') \in oo \wedge (o', sign'') \in osign \} \} \end{array} \right\}$$

SQL join signatures can include several semantically identical columns. For instance, the following SQL query has two semantically identical columns “Person_ID”:

```

SELECT * FROM PersonTaskRel ptr, PersonProjectRel ppr
WHERE ptr.Person_ID = ppr.Person_ID AND ppr.Project_ID = 'MES',

```

while the equivalent composition “(Task–Person–Project=“MES”)” has only one column for the concept “Person” in spite of the fact that both composed associations contain the concept.

Another important property of composition operations is implicit join predicates, while, for example, SQL requires definition of join predicates in the explicit form. Composition join predicates are constructed automatically as equality of identical role concepts of different nested operations. For instance, the associations (Employee, Person), (Person, Phone), and (Person, Skill Level, Skill Type) were composed by the concept “Person” without an explicit predicate. Note that the query has the result

signature “Employee, Person, Phone, Skill Level, Skill Type” with the single concept “Person.”

An SCQL composition operation can be outer one. In this case, all nested operations are divided into two categories: outer and non-outer. Such composition operation is executed in two stages: a) a non-outer composition operation of all non-outer nested operations is first fulfilled; and b) the result of the non-outer composition is then extended with all compatible instances of outer operations. The extension procedure is the following. Two instances are considered to be compatible if they have the same values for all common role concepts. Select an instance of the non-outer composition and all its compatible instances of outer-nested operations. Make a partial composition of the non-outer composition and the selected outer operations, taking into account only the selected instances. Repeating such partial composition for all instance of the non-outer composition, one creates the extension that is the result of the desired outer composition. The outer composition operation type is analogous to the SQL outer join, but it is simpler and more transparent owing to the same reasons as the non-outer composition operation type.

Outer-nested operations are marked with the plus sign right after, and a composition is outer one if it has at least one outer-nested operation. For instance, the query “((Person, Task)+, (Person–Age<30))” is the outer composition of the subquery (Person–Age<30) (see this section below for details) and the association (Person, Task). The former subquery is extended using the latter one, which is marked with the plus sign. The analogous SQL query is:

```
SELECT ptr.Task_ID, ptr.Person_ID, p.Age FROM Person p, PersonTaskRel ptr
WHERE p.Person_ID = ptr.Person_ID(+) AND p.Age < 30.
```

One can see that the SQL query uses the outer join and is more complicated than the analogous SCQL query.

Formally, the outer composition operation set $ocomp$ is a subset of the general composition operation set: $ocomp \subseteq comp$; and the outer operation set $outo$ is a subset of the general operation set: $outo \subseteq o$. There are several constraints imposed on outer compositions and their nested operations. First of all, each outer composition operation is to have at least one non-outer nested operation:

$$[C9] \forall ocomp' \in ocomp \exists o' \in o \{ (o', ocomp') \in oo \wedge o' \notin outo \}$$

All outer operations nested to an outer composition operation are to have common role concepts with non-outer ones:

$$[C10] \forall a \in outo \exists oc \in ocomp \exists b \in o \{ (a, oc) \in oo \wedge (b, oc) \in oo \wedge b \notin outo \wedge directly_connected(a, b) \}$$

where the *directly_connected* predicate is as follows:

$$directly_connected(a \in o, b \in o) \equiv \exists sign_a \in sign \exists sign_b \in sign \exists oc \in comp \\ \{ (a, oc) \in oo \wedge (b, oc) \in oo \wedge (a, sign_a) \in osign \wedge (b, sign_b) \in osign \wedge sign_a \cap sign_b \neq \emptyset \}$$

The last outer composition constraint is the following: all non-outer nested operations are to form a connected graph:

$$\forall oc \in ocomp \forall a \in o \forall b \in o$$

$$[C11] \{(a, oc) \in oo \wedge (b, oc) \in oo \wedge a \notin outho \wedge b \notin outho \rightarrow connected(a, b)\}$$

where the *connected* predicate is as follows:

$$connected(a \in o, b \in o) \equiv directly_connected(a, b) \vee \exists d \in o \{directly_connected(a, d) \wedge connected(d, b)\}$$

Also, the composition serves for filtering subqueries. For this purpose, a composition operation can have some nested operations based on logical predicates defined over role concepts. Such operations are named as logical selections. A logical selection is a leaf operation selecting all instances (combinations of role concept values) satisfying a given predicate. Consider a composition operation based on some logical selections and other operations. Derive another composition (non-filtered one) based on all nested operations of the desired composition besides logical selections. The desired composition's result contains only those instances of the non-filtered composition's result that satisfy all predicates of all the nested logical selections. So composition of logical selections and other operations results in additional filtering effect—only those instances that satisfy all predicates of the nested logical selections are kept.

Logical selections are written as logical predicates in round brackets. For instance, the query ((Task-Person-Age), (Age<30)) has the logical selection (Age<30) that contains all ages less than 30; the same query in the short notation is written as (Task-Person-Age<30). The query in both notations reads as “select all tasks being solved by persons younger than 30.” The analogous SQL query is:

```
SELECT ptr.Task_ID, ptr.Person_ID, p.Age FROM Person p, PersonTaskRel ptr
WHERE p.Person_ID = ptr.Person_ID AND p.Age < 30.
```

As one can see, the SCQL query is a lot more simple and transparent than the analogous SQL query, as the SCQL query does not apply proper association names, explicit join predicates, and has semantic signature as opposed to the SQL query.

Formally, let p be a set of predicates based on role concepts, ls be a logical selection set being subset of the general leaf operation set: $ls \subset lo$. Then each logical selection must be characterized by one predicate: $lsp : ls \rightarrow p$,

$$[C12] \forall ls' \in ls \exists p' \in p \{(ls', p') \in lsp\}$$

Any logical selection is nested to a composition operation:

$$[C13] \forall ls' \in ls \exists o' \in o \{(ls', o') \in oo \wedge o' \in comp\}$$

The logical selection is to be based on role concepts of other operations nested to the same composition and not being logical selections:

$$[C14] \quad \forall c \in comp \forall ls' \in ls \{signature(ls') \subseteq \bigcup \{signature(o') \mid (o', c) \in oo \wedge o' \notin ls\}\}$$

where the *signature* function is as follows:

$$signature(a \in o) \equiv \{sign' \mid (a, sign') \in osign\}$$

Another leaf operation type is the association selection. An operation of this type is defined over a role concept set being its signature and is executed in either of two ways: a) it selects all instances of an association based on the given concept set if the queried schema contains such association; or b) it executes, using the current context, a shortened query that is a selection of indirect interrelations of the given concept set. A signature being a role concept set is the structure and the only parameter of the association selection operation.

The association selection was discussed in the previous section as two ways of referring to associations: directly and indirectly using contexts. The notation of the operation type is enumeration of role concepts in round brackets. For example, the enumeration (Person, Phone) is the direct selection of the appropriate association, and the enumeration (Employee, Phone) is the indirect selection of the concepts' interrelation by means of a composition-projection shortened query uncovering to (Employee–Person–Phone).(Employee, Phone) (see the projection syntax below) using the current context of the running example. See the previous section for details concerning both ways of selecting concept set interrelation.

One can see that a SCQL association selection does not use association (relation) proper names, which results in more simplicity and transparency of SCQL queries in comparison with other query languages. Formally, let *as* be an association selection set being subset of the general leaf operation set: $as \subset lo$. It is true that all leaf operations of SCQL expressions are to be either association selections or logical selections:

$$[C15] \quad \forall lo' \in lo \forall e' \in scql \{(lo', e') \in oe \rightarrow lo' \in ls \vee lo' \in as\}$$

where the set *scql* is a subset of the general expression set: $scql \subseteq e$.

The next non-leaf SCQL operation type is the minus. An operation of this type is based on two ordered nested operations. The result of the operation consists of all instances of the first nested operation that have no compatible instances among instances of the second nested operation (see the definition of compatible instances above). The SCQL minus is more simple than the analogous operation types of other query languages since it does not require alignment of nested operation signatures. For instance, selection of phones of persons having no “experienced” skills can be written down in SQL as:

```
SELECT Person_ID, Phone FROM Person
MINUS SELECT p.Person_ID, p.Phone FROM Person p, Skill s
WHERE p.Person_ID = s.Person_ID AND s.Level = 1 /*Experienced*/.
```

The analogous SCQL query can be written as ((Person, Phone) minus (Skill Level="Experienced," Skill Type, Person)). One can see that the second nested operation's signature is not aligned to the first one as opposed to the above SQL query. The resulting signature of the minus operation is the signature of the first nested operation (Person, Phone), in spite of the fact that the second nested operation has an absolutely different signature.

Saying this formally, a minus operation has exactly two nested operations placed at one of two positions: $p \equiv \{1, 2\}$, $op : o \rightarrow p$,

[C16]

$$\forall m \in \text{minus} \exists_1 no_1 \in o \exists_1 no_2 \in o \left\{ \begin{array}{l} no_1 \neq no_2 \wedge (no_1, m) \in oo \wedge (no_2, m) \in oo \wedge (no_1, 1) \in op \wedge \\ (no_2, 2) \in op \wedge \{no \mid (no, m) \in oo\} = \{no_1, no_2\} \end{array} \right\}$$

The result signature of any minus operation is a signature of its first nested operation:

[C17]

$$\forall m \in \text{minus} \forall no \in o \{ (no, m) \in oo \wedge (no, 1) \in op \rightarrow \text{signature}(m) = \text{signature}(no) \}$$

The next non-leaf SCQL operation type is the union. Operations of this type have an unordered set of nested operations. The result of a union operation is calculated as follows. Select an instance of a nested operation and all its compatible instances of other nested operations (see above for instance compatibility definition). Then extend the selected instances with all other instances compatible with at least one already selected instance (the compatible instances are to be of different operations). Repeat the extension procedure while the set of selected instances is expanded. The resulting set of selected instances is a composition cluster. Calculate all existing composition clusters using all the nested operations. For each composition cluster, calculate a partial composition of nested operations included with the cluster, taking into account only those instances that are in the cluster. If the cluster includes an only nested operation, the partial composition has resulted in only that instance that is included with the cluster. The result of the union is all instances of all the partial compositions.

One can see that the union is more extensive than analogous operation types of other query languages since it does not require alignment of signatures of nested operations. Both the full outer join and the union of SQL are special cases of the SCQL union: a) if nested operations have the same signature, the SCQL union is analogue for the SQL union; b) if there are two nested operations with different signatures, the SCQL union is analogous to the SQL full outer join; or c) otherwise, SQL (and other languages) has no analogues for the SCQL union.

The SCQL union has the intuitively clear notation: the word “union” between each pair of subqueries to be united. For instance, the query “for each project select all persons being its members or solving its tasks” is written down as ((Project–Person) union (Project–Task–Person)).(Project, Person)). The analogous SQL query is as follows:

```
SELECT e.Person_ID, ppr.Project_ID FROM Employee e, PersonProjectRel ppr
WHERE e.Person_ID = ppr.Person_ID
UNION SELECT e.Person_ID, t.Project_ID
FROM Employee e, Task t, PersonTaskRel ptr
WHERE e.Person_ID = ptr.Person_ID AND ptr.Task_ID = t.Task_ID.
```

It is important that SCQL union operations have simple structure since the nested operations are not to be reorganized to have the equal signatures. A union operation is executed taking into account application domain concepts completely and, as a result, it has a semantic signature being union of signatures of all nested operations:

$$[C18] \quad \forall u \in \text{union} \{ \text{signature}(u) = \{ \text{signature}(no) \mid (no, u) \in oo \} \}$$

The last non-leaf SCQL operation is the transformation. Signatures of all operations described above are completely calculated from signatures of nested operations. Transformation operations cover all cases concerned with controlled modification of a single nested operation’s signature by dint of projection, grouping, and calculation of both aggregate and non-aggregate functions:

$$[C19] \quad \forall t \in \text{trans} \exists no \{ (no, t) \in oo \}$$

One defines a transformation operation by specifying a resulting signature on basis of a nested operation signature. Role concepts of the resulting signature can be of the following types: projected, calculated, and aggregated. A projected role concept is a role concept of the nested operation copied from it without any modification. A calculated role concept is a role concept that is calculated with a non-aggregate function based on role concepts of the nested operation. And an aggregated role concept is a role concept calculated with an aggregate function based on the nested operation’s role concepts that are not used as a projected role concept or for computation of a calculated role concept.

A transformation operation is calculated in the following way. Consider a transformation operation and its nested operation. For each initial instance of the nested operation, calculate a derived instance, filling it with a) the copied values of projected role concepts, and b) values of calculated role concepts computed with the specified functions from the initial instance. If the transformation operation has no aggregated role concepts, then its result is all distinct derived instances (two instances are considered to be equal if they have the same set of pairs <a role concept, its value>). Otherwise, grouping is necessary to calculate the aggregate role concepts: group initial instances by equal derived instances. For each resulting group extend its derived instance with aggregated role concepts computed using the specified aggregate functions from all initial instances of the group. In this case, the result of the transformation operation is

all the extended derived instances. One can see that a transformation operation can fulfill all the modification types: projection, calculation, and grouping. The transformation permits almost any combinations of the modification types. One can prove that any transformation operation is to have at least one role concept not being aggregated.

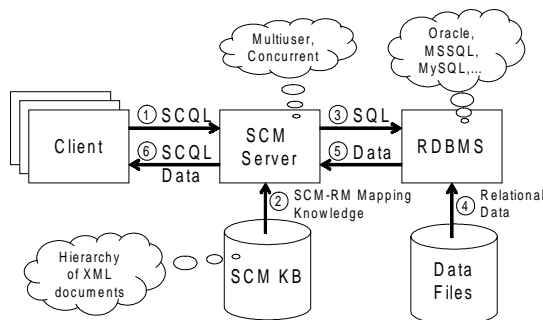
SCQL provides for two alternative notations of resulting signature definition—in round brackets after dot, and between ‘SELECT ... FROM’ words. Both ways are equivalent, but the first one is used after nested operation expression, and the second one is used before it. The way to use is determined by a user’s preferences. SCQL does not require specifying role concepts to be grouped in an explicit form. Grouping is done implicitly by role concepts not used for computing aggregate functions. For instance, the average age of persons working on a project can be calculated as (Project–Person–Age).(Project, AVG(Age)), or as (SELECT Project, AVG(Age) FROM (Project–Person–Age)). Here grouping on the concept “Project” is implicitly done because the aggregate function AVG is used. Using the context mechanism, the same query can be written down as (Project, AVG(Age)), which is very simple and transparent. The analogous SQL query is the following:

```
SELECT p.Person_ID, AVG(p.Age) FROM PersonProjectRel ppr, Person p
WHERE ppr.Person_ID = p.Person_ID GROUP BY ppr.Project_ID.
```

One can see that the SQL query uses proper table names, an explicit join criterion, a grouping clause, and has no semantic signature, unlike the analogous SCQL query.

Table 2. Summary of SCQL operations’ structure

Figure 5. SCM-based client/server technology architecture



Note that signatures of all SCQL operations are unordered sets of role concepts. Signatures of all non-leaf operations, besides transformation operations, are calculated on the basis of nested operations' signatures. Only signatures of transformation operation, logical selection, and association selection operations are to be specified explicitly (see the summary of SCQL operations' structure in Table 2).

APPLICATIONS AND FUTURE TRENDS

Now SCQL is used as the foundation of SCM-based client/server technology that permits client programs to communicate with SCM servers in terms of SCQL queries. At the moment, SCM servers are backed by any existing relational database management system (RDBMS), and it will support other DBMS types in the future. The technology permits creating client/server applications interacting with end-users in terms of application domain concepts and their unique associations. Any existing RDBMS of a given version or higher can now be overbuilt with an SCM server to create SCM-based client/server applications (Ovchinnikov, 2005a).

The technology works as illustrated in Figure 5. First of all, one or several clients send some SCQL queries to an SCM server concurrently. The server processes each query in a separate thread. During processing, it uses an SCM knowledge base that is a hierarchy of XML documents describing mapping of existing relational schema to a published SCM schema. The processing results are SQL queries transmitted to an RDBMS. The RDBMS executes each SQL query and returns results back to the SCM server. The SCM server returns query execution results to appropriate clients in the SCQL form. Within the technology, clients are abstracted from data storing and can execute queries in terms of an application domain by dint of SCQL.

The most emerging direction of development of the technology is creation of a data integration system that permits fusion of several heterogeneous SCM servers into one space. Any client is expected to use the space the same way as a single SCM server. The first sub-direction is finding the ways of distributed SCQL queries' execution within a heterogeneous environment of SCM servers. The second sub-direction is application of the transaction control theory to information modification through the unified SCM

interface. And another sub-direction is elaboration of methods of mapping SCM schemas to schemas of other types and SCQL queries to queries of other query languages.

The other important development trend is creation of the tool “semantic browser.” The tool is expected to solve some different tasks. The first task is building GUI applications by dint of semantic-oriented components that publish its data as a derivation of a SCM schema using application domain concepts. The applications are expected to be created using mostly declarative way, and so to be suitable for end-users who are not IT specialists. The second task is integration of all information representation forms to one space with built-in support of semantic navigation between the existing forms and applications. For instance, one method of semantic navigation is the following. One selects values of a concept and requests the semantic browser to create a list of all possible transitions from the concept values. The browser analyses all existing forms, selects those to which one can move in the context of the concept, and gives the resulting list. One selects a desired form, the browser loads the form and adjusts it to show information about the selected concept values only. As a result, one has made a transition, which was not programmed in advance, within the selected concept values. A set of possible transitions is completely determined by a set of existing forms and their structures. So Semantic Browser is expected to permit semantic navigation over structured data published in the form of a SCM schema.

And the last perspective trend to be mentioned is the use of SCM as a computational independent model (CIM) of OMG model-driven architecture (OMG, 2003a). Since SCM is the most abstract model having the less possible set of implementation detail expressive means, the author believes it best suits the CIM purpose and can fill the gap of CIM models in MDA. The trend has a mass of sub-trends directed at implementation of tools serving for automation of using SCM as CIM (for instance, mapping SCM to other models, including UML (OMG, 2003b, 2004).

CONCLUSION

The Semantically Complete Query Language (SCQL) is a declarative conceptual query language not using proper relation names, based on the semantically complete model (SCM). All non-leaf operations of SCQL, besides transformations, are merely parameterized by nested operation sets and have no any additional parameters; and transformations have the only additional parameter that is a resulting signature. Signatures of all SCQL operations are unordered. Composition operations do not require join predicates in an explicit form. As a result, SCQL is more semantically transparent and simple for end-users than other conceptual and data query languages since the latter require more detailed specification of the query execution way: proper relation names, explicit join predicates, grouping fields, and other details.

As opposed to other query languages, signatures of all SCQL queries are semantic as they are sets of application domain concepts and not sequences of abstract columns not associated with concepts of an underlying application domain. Additionally, SCQL does not permit creation of some type of senseless queries. Path and star expressions are written using very intuitively clear notations.

SCQL has introduced a context mechanism that permits querying some indirectly associated concepts as if they are associated directly, adopting SCQL queries to users’

Table 3. Summary of properties of SCQL expressions

Characteristic	Example
Application domain concept usage	(Project, Task)
Semantic nature of resulting signatures	(Person, Phone)
Concept-based query formulation (Uselessness of proper relation names)	(Person, Skill Level, Skill Type)
Capability of implicit join predicates	((Project, Task), (Task, Person))
Prohibition of senseless queries	SQL: ... WHERE a.Project_ID = b.Task_ID
Capability of queries as concept chains and stars	(Project–Task–Person) or (Task–[Project, Person])
Capability of join-like queries formulated as resulting signatures merely	Using the current context: (Employee, Phone)
Some query adaptation without rewriting	Different meaning in different contexts: (Employee, Project)

needs without query rewriting, and browsing a SCM schema by end-users in the more transparent way. As a result, SCQL is useful both for end-users and for IT professionals. All the properties are summarized in Table 3.

All the described properties make SCQL a truly semantic query language that is founded on application domain concepts. Its queries are easy-to-read, write, check, and translate to a natural language, and they are not concerned with implementation details at all. Other existing query languages share some of the properties to a certain extent, but none of them has all the properties in full measure.

REFERENCES

- Bakema, G., Zwart, J., & van der Lek, H. (1994). Fully communication oriented NIAM. In G. M. Nijssen, & J. Sharp (Eds.), *NIAM-ISDM 1994 Conference. Working Papers*, (pp. L1-35).

- Bloesch, A.C. & Halpin, T.A. (1997). Conceptual queries using ConQuer-II. In *Proceedings of ER '97: 16-th International Conference on Conceptual Modeling*, (pp. 113-126).
- van Bommel, P., ter Hofstede, A.H.M., & van der Weide, Th.P. (1991). Semantics and verification of object-role models. *Information Systems*, 16(5), 471-495.
- Bronts, G.H.W.M., Brouwer, S.J., Martens, C.L.J., & Proper, H.A. (1995). A unifying object role modelling approach. *Information Systems*, 20(3), 213-235.
- Brouwer, S.J., Martens, C.L.J., Bronts, G.H.W.M., & Proper, H.A. (1994). Towards a unifying object role modelling approach. In *Proceedings of the First International Conference on Object-Role Modelling (ORM-1)*, (pp. 259-273).
- Chen, P.P.S. (1976). The entity-relationship model – Towards a unified view of data. *ACM Transactions on Database Systems*, 1(1), 9-36.
- Chen, P.P.S. (1981, October 12-14). A preliminary framework for entity-relationship models. In P.P. Chen (Ed.), *Proceedings of 2nd International Conference on Entity-Relationship Approach to Information Modeling and Analysis*, (pp. 19-28). North-Holland.
- Codd, E.F. (1972). *Relational completeness of data base sublanguages*. *Data Base Systems, Courant Computer Science Symposia Series 6*. Upper Saddle River, NJ: Prentice-Hall.
- Codd, E.F. (1979). Extending the database relational model to capture more meaning. *ACM Transactions on Database Systems*, 4(4), 397-434.
- Dibie-Barthelemy, J., Haemmerle, O., & Loiseau, S. (2001). Refinement of conceptual graphs. *Vol. 2120 of LNAI*, (p. 216).
- Embley, D.W., Wu, H.A., Pinkston, J.S., & Czejdo, B. (1996). *OSM-QL: A calculus-based graphical query language*. Technical Report, Dept. of Computer Science. Brigham Young University, Salt Lake City, Utah.
- Halpin, T.A. (1995). *Conceptual schema and relational database design*. Sydney, AUS: Prentice-Hall.
- Halpin, T.A. (2001). *Information modeling and relational databases*. San Francisco, CA: Morgan Kaufmann.
- Halpin, T.A. (2004). Business rule verbalization. In A. Doroshenko, T. Halpin, S. Liddle, H. Mayr (Eds.), *Proceedings of the 3rd International Conference ISTA '2004: Information Systems Technology and its Applications*, (pp. 39-52). Salt Lake City, UT: GI Lecture Notes in Informatics P-48.
- Halpin, T.A. & Orłowska, M.E. (1992). Fact-oriented modelling for data analysis. *Journal of Information Systems*, 2(2), 1-23.
- ter Hofstede, A.H.M., Proper, H.A., & van der Weide. (1993). Formal definition of a conceptual language for the description and manipulation of information models. *Information Systems*, 18(7), 489-523.
- ter Hofstede, A.H.M., Proper, H.A., & van der Weide, Th.P. (1996). Query formulation as an information retrieval problem. *The Computer Journal*, 39, 255-274.
- ter Hofstede, A.H.M., Proper, H.A., & van der Weide, Th.P. (1997). Exploiting fact verbalisation in conceptual information modelling. *Information Systems*, 22(6/7), 349-385.
- ter Hofstede, A.H.M., & van der Weide, Th.P. (1993). Expressiveness in conceptual data modeling. *Data & Knowledge Engineering*, 10(1), 65-100.
- Nijssen, G.M. & Halpin, T.A. (1989). *Conceptual Schema and Relational Database Design: a fact oriented approach*. Sydney: Prentice-Hall.

- OMG (2003a). *OMG Model Driven Architecture Guide V1.0.1.*, June 1. [Online]. Retrieved from <http://www.omg.org/cgi-bin/doc?omg/03-06-01>
- OMG (2003a). *OMG UML 2.0 Infrastructure Specification*, September 15. [Online]. Retrieved from <http://www.omg.org/cgi-bin/doc?ptc/2003-09-15>
- OMG (2004). *OMG UML 2.0 Superstructure Specification*. [Online]. Retrieved from <http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>
- Ovchinnikov, V.V. (2004a). A conceptual modeling technique based on semantically complete model, its applications. In A. Doroshenko, T. Halpin, S. Liddle, & H. Mayr (Eds.), *Proceedings of the 3rd International Conference ISTA'2004: Information Systems Technology and its Applications*, (pp. 25-38). Salt Lake City, UT: GI Lecture Notes in Informatics P-48.
- Ovchinnikov, V.V. (2004b). A semantically complete conceptual modeling technique. *Journal of Conceptual Modeling*, 32. [Online]. Retrieved from <http://www.inconcept.com/jcm>
- Ovchinnikov, V.V. (2004c). Improving controllability of vast conceptual models. *Journal of Conceptual Modeling*, 31. [Online]. Retrieved from <http://www.inconcept.com/jcm>
- Ovchinnikov, V.V. (2005a). *SCM portal*. [Online]. Retrieved from <http://scm.lipetsk.ru>
- Ovchinnikov, V.V. (2005b). A concept-based query language not using proper relation names. In J. Castro, & E. Teniente (Eds.), *Proceedings of CAiSE'05 Workshops, Vol. 1*, (pp. 617-628). Porto: FEUP.
- Ovchinnikov, V.V., & Vahromeev, Y.V. (2005). A declarative concept-based query language as a mean for relational database querying. *Journal of Conceptual Modeling*, 34. [Online]. Retrieved from <http://www.inconcept.com/jcm>
- Owei, V. (2000). Natural language querying of databases: An information extraction approach in the conceptual query language. *International Journal of Human-Computer Studies*, 53(4), 439-492.
- Owei, V., & Navathe, S. (2001a). A formal basis for an abbreviated concept-based query language. *Data & Knowledge Engineering*, 36(2), 109-151.
- Owei, V., & Navathe, S. (2001b). Enriching the conceptual basis for query formulation through relationship semantics in databases. *Information Systems*, 26(6), 445-475.
- Owei, V., Navathe, S.B., & Rhee, H.-S. (2002). An abbreviated concept-based query language and its exploratory evaluation. *Journal of Systems and Software*, 63(1), 45-67.
- Seaborne, A. (2004). RDQL - A query language for RDF. *W3C Member Submission*. [Online]. Retrieved from <http://www.w3.org/Submission/2004/SUBM-RDQL-20040109/>
- de Troyer, O.M.F. (1991). The OO-binary relationship model: A truly object-oriented conceptual model. In *Proceedings of the Third International Conference CaiSE'91 on Advanced Information Systems Engineering, Vol. 498 of LNCS*, (pp. 561-578). Berlin: Springer-Verlag.
- W3C (2004a). Resource Description Framework (RDF): Concepts and abstract Syntax. In G. Klyne, & J. J. Carroll (Eds.), *W3C Recommendation*. [Online]. Retrieved from <http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/>
- W3C (2004b). OWL Web Ontology Language Reference. In M. Dean, & G. Schreiber (Eds.), *W3C Recommendation*. [Online]. Retrieved from <http://www.w3.org/TR/2004/REC-owl-ref-20040210/>

ENDNOTES

- ¹ A signature implies a sequence or a set, depending on a language, of pairs <a result column, its value type>. In the context of SCQL, a signature is a set of domain concepts.