A Concept-Based Query Language Not Using Proper Relation Names

Vladimir Ovchinnikov, PhD in Computer Science

Russia, 398024, Lipetsk, prospekt Pobedi 104-44,
ovch@lipetsk.ru,
Lipetsk State Technical University, Russia,
Novolipetsk Iron & Steel Corporation, Russia.

Abstract. The paper is focused on a concept-based query language that permits querying by using only application domain concepts. The query language has features making it simple and transparent for end-users: each query operation is completely defined by its result signature and nested operation's signatures; a query's signature represents an unordered set of application domain concepts; join predicates are not to be specified in an explicit form. In addition, the paper introduces constructions of closures and contexts as applied to the language. The constructions 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. This query language is named as SCQL (Semantically Complete Query Language).

Keywords. Database Modeling and Query Languages, Conceptual Query Language, Concept-Based Query Language, Semantically Complete Query Language.

1. Introduction

Conceptual models serve as means of application domain modeling as opposed to means of system implementation modeling. A conceptual model does not concern with implementation details and represents a description of an application domain's essence. Conceptual query languages are meant as tools for querying conceptual models and have dual use. On the one hand, conceptual queries play the key role in constraint formalization: any constraint can be represented as an underlying query and an assertion upon it. On the other hand, such queries can be used for data request formulation. In both cases, conceptual query transparency and simplicity are the most important.

Conceptual query simplification can be achieved by using natural names for entities and relations when modeling and querying [4, 7, 10, 14]. The method significantly simplifies end-users' work as interaction with systems takes place directly in application domains' terms. Examples of such query languages are LISA-D [5, 6] based on Object-Role Modeling [3], CQL [10]. But this simplification

is not structural: the languages' core 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 which are assigned to persons; also persons can directly participate in projects' teams. The domain has the following formalization in ORM:

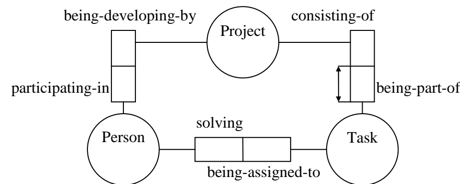

Fig. 1. ORM Model of Project Management Domain

The query "select all tasks assigned to persons participating in the project MES" 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 relations is important and should be kept correct; b) the appropriate predicator precise name should be remembered. A user is not insured against creation of senseless queries like "Task solving Person", "Person consisting of Project", or mistakes like "Person solved Task". In addition, LISA-D does not support subqueries [5]. Using the language proposed further, the query is formulated as (Task–Person–Project="MES"). A user has not to remember relation or predicator precise names and has not to keep correct sequence of relations. Also the language does support for subqueries.

Unfortunately, LISA-D did not become an industrial standard for information system development and is slightly supported by tools. Now the standard is SQL. Therefore, let us use SQL as comparison basis below. The above example domain can be formalized as ER and physical models as the Fig. 2 shows. To use the example below as running one, we introduce person's phone, age, skills, and employees as persons' particular case.

The above query should be 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'
```

In comparison with the query (Task–Person–Project="MES") proposed, the SQL query has the following complexities: a) the join predicate "ptr.Person_ID = ppr.Person_ID" is defined explicitly; b) the appropriate precise table names should be remembered; c) the query's result signature is lacking in semantics since it consists of abstract columns not associated with the domain's concepts; d) fields' and tables' names are far from natural language.

LISA-D, SQL, and other query languages use proper relation names for referencing to relations. A user has to remember a lot of precise names to formulate queries. The reason of this lies in underlying modeling techniques: ORM, ER, and others. The techniques imply identification of relations by their proper names. Any two entities of the models can have a lot of association ways, and each of the ways

takes its own unique name. Not all relations can be named clearly and shortly. Sometimes full names of relations are whole sentences, actually enumerating participated concepts and only. For instance, the relationship of “Person” and “Task” can be named like “Persons solving tasks”, “Tasks being solved by persons”, or “Assignments of tasks to persons”. A user cannot think about entities as they are merely associated. But it is not necessary to remember the association’s precise name if one refers to it by (Person, Task) as the proposed language implies. Another complexity factor is that queries’ result signatures are not based on application domains’ concepts, and a query’s result interpretation is completely determined by its structure. For instance, the result “Task_ID” in the previous SQL query can mean anything a priori, even phone number. Moreover, a user is not insured against formulation of senseless queries, for instance, joining tables with “Age = Level”. As a result, the languages are too complicated for end-users.

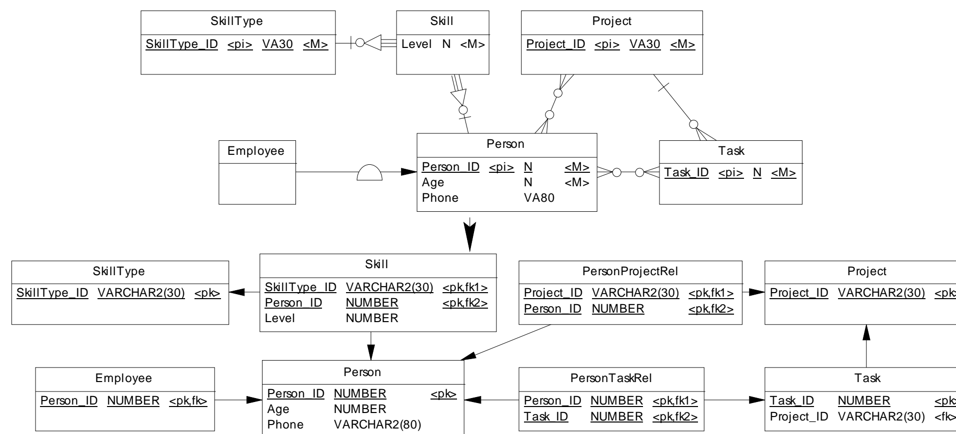

Fig. 2. ER and Physical Model of Project Management Domain

Another way of query simplification is use of a GUI application concealing query complexity, as, for instance, Conquer-II [1] and OSM-QL [2] propose. Using intuitively clear interface elements like trees, an end-user can easily construct conceptual queries. Nevertheless, the way has the limit imposed by strong impact of a query language’s core on a user interface structure, for example, tree node types, node connectivity and node attributes. Since each operation of the language proposed further is completely defined by its resulting signature and nested operation’s signatures, we suppose that the language suites the purpose well and should be developed in the direction in the future.

2. Restriction Imposed on Underlying Model

It is known the only way of referencing to relations without explicit naming it: to use concept combinations as references to relations so that each concept combination gets identification of an appropriate relation. The identification would imply a sequence of

concepts, but this method is acknowledged as non-transparent. The proposed language uses another way of relation identification: by means of application domain concept sets. Not any model can be used as basis for a query language not using proper relation names. Such model is to permit identification of relations by domain concept sets. The model having the property was proposed at [11, 13] and was named as Semantically Complete Model (SCM). SCM is a full-scale modeling technique that is near to natural language text (see [11, 13] for notation details). SCM model of the above running example is the following:

Person solves Tasks
Person has a Skill Level for a Skill Type
  [(Person, Skill Type) → Skill Level]
Person is of Age →
Person has a Phone →
Employee is a Person ≡
Project consists of Tasks ←
Project has a team of Persons

Formally, an SCM model represents a set of associations based on sets of concepts. Let m be set of all SCM models, a be set of all associations, c be set of all concepts, $ma \subseteq m \times a$ determine associations for models, and $ac \subseteq a \times c$ determine concepts for associations. Then the association identification constraint "a model cannot have two associations based on the same set of concepts" is formulated as follows:

[C1] $\forall m' \in m \forall a' \in a \forall a'' \in a \left\{ \begin{array}{l} (m',a') \in ma \wedge (m',a'') \in ma \rightarrow \\ \{c' \mid (a',c') \in ac\} \neq \{c' \mid (a'',c') \in ac\} \end{array} \right\}$.

SCM has another restriction proving its name: an association of one SCM model cannot be based on a concept set being proper subset of other associations' concept sets. This restriction guarantees each association defines *semantics* of underlying concepts' correlation *completely*.

[C2] $\forall m' \in m \forall a' \in a \forall a'' \in a \left\{ \begin{array}{l} (m',a') \in ma \wedge (m',a'') \in ma \rightarrow \\ \{c' \mid (a',c') \in ac\} \not\subset \{c' \mid (a'',c') \in ac\} \end{array} \right\}$.

The restriction is not discussed in the paper in details as it does not influence on main properties of the language proposed (see [12] for details). The conceptual language based on SCM and not using proper association names for query formulation is named as Semantically Complete Query Language (SCQL) [12, 13].

3. Association Referencing within SCQL Expressions

SCQL does not use proper association names. As a result of C1, defining a concept set, one makes a reference to an association. SCQL provides for simple notation of such references: enumeration of concepts by comma in round brackets. Concept order in such enumerations is not important. For instance, both of the references (Person, Skill Level, Skill Type) and (Person, Skill Type, Skill Level) are correct. As you see, using SCQL, one has not to remember proper association names and should know the fact of concepts' correlation only.

A SCQL composition can be considered as a mathematical composition of associations based on domain concepts and not sets (see the next section for details). A SCQL path expression is a composition of several binary associations and is

defined as concept chains. Each chained concept pair is considered as a concept set referencing to an association. A chain as a whole represents a composition of all associations referenced by pairs indicated with dashes. For instance, (Person–Task–Project) selects for each task its project and assigned persons by composing the associations (Person, Task) and (Task, Project). As you see, chain links 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 defined explicitly. SCQL path expressions can be written starting from any edge concept, for example, (Project–Task–Person) is equivalent to (Person–Task–Project). A star expression, when binary associations of a composition are organized as a star with one central concept, can be formulated using the path notation as (Phone–Person–Project), or using the star notation as (Person–[Project, Phone, …]).

One concept can play several roles within an expression, for example, when using one association several times. For this purpose SCQL has the conception "role concept". A role concept represents a concept extended with a role name indicating the concept's role in a certain SCQL 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 the projects are semantically diverse columns of the expression's result and have the role names "Task's" and "Person's". Not using the roles, the expression becomes cyclic with one project column: (Project–Task–Person–Project), which reads as "select persons with their tasks being part of projects 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). Formally, let rc be set of role concepts, rn be set of role names. Then the maps $rcc: rc \to c$ and $rcrn: rc \to rn$ reflect that each role concept pertains to a concept and can have a role name.

SCQL expressions can be additionally simplified by using association closures. An association closure is defined over one SCM model and represents a set of associations. When applying a closure, some indirectly associated concepts can be used within expressions as if they are connected directly. For example, if (Employee, Person) and (Person, Phone) are included to a closure, one can select data using (Employee, Phone) or (Employee–Phone) instead of (Employee–Person–Phone). Closures increase query transparency since they permit omitting transition details when it is obvious for users. Actually, a closure represents a predefined composition being used implicitly, namely a composition of all associations included to the closure. SCQL provides for transparent way of closure use: by enumerating concepts (role concepts) in round brackets, one can make a reference to an association as well as a projection of a closure. For a given concept set a closure is applied only if there is no an association based on it. For example, (Employee, Phone) is projection of the closure's composition (Employee–Person–Phone), while (Person, Phone) is reference to the appropriate association. Formally, let ac be set of all closures. Then $aca \subseteq ac \times a$ defines associations included to each closure. Closures serve for similar purpose as the abbreviated concept-based query language presented in [8, 9] which does not require entire query paths to be specified but only their terminal points.

Closures make possible governing expression execution contexts. The simplest context is empty one when closures are not used and all concept enumerations refer to associations directly as (Person, Phone). Contexts can contain several active closures.

In this case, all associations of active closures are united to one consolidated closure of a context. If a concept enumeration does not refer to an association, the consolidated closure is projected on these concepts. A special case of closures is default closures. A default closure of a model is used by default for expressions if the contrary is not indicated explicitly. A closure not being default is to be uniquely named. Closures can be added and removed explicitly when tuning an expression execution context. Contexts permit simplifying and clarifying query expressions significantly. For instance, all non-cycle path and star expressions can by written as simple enumeration of required concepts without indicating chains or stars explicitly as (Employee, Phone) in the example above. Formally, let cx be context set, then $cxac \subseteq cx \times ac$ defines closures constituting each context. Both default closures dac and named closures ncc are subsets of the general closure set: $dac \subseteq ac$, $nac \subseteq ac$.

Query semantics can change greatly when modifying a context. It makes SCQL more flexible, but if one uses the context mechanism heedlessly, semantics of complex queries can change unpredictably; context change is to be closely controlled. Note that all SCQL queries stay sensible in any context, but senses differ. SCQL contexts 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, contexts can be changed explicitly or automatically by using browsing statistics, for instance, a frequency of association use. Another strategy of context creation aims to reflection of shortenings generally accepted by a community or an application domain. The shortenings underlie the default closure. Additionally, several named closures can be created to provide for some optional shortening being active for some part of a community or an application domain. The options are activated when necessary. 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 previous text part is used as basis for a context of next text part and the last context is modified by using some statistics of both text parts. Context mechanism of SCQL is unique in its own. Other known languages have no context mechanisms at such deep architectural level: context change does not require query rewriting.

4. SCQL expression properties

The paper is focused on the following main SCQL property: it permits query formulation only by using application domain concepts. It is guaranteed by the fact that associations are identified by concept sets. The property increases transparency and simplicity of query expressions, especially when using closures, path and star expressions. Furthermore, SCQL has other interesting properties based on features 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 are to be selections from relations of an underlying model. Let e be set of all expressions, o be set of operations, ot be set of operation types, $oe : o \rightarrow e$ determine operations

of each expression, and $oot: o \to ot$ determine an operation type for each operation. Then 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:

[C3] $\forall o' \in o \forall ot' \in ot \left\{ \begin{array}{l} (o', ot') \in oot \to \\ \{(o' \in lo \wedge ot' \in lot) \vee (o' \notin lo \wedge ot' \notin lot)\} \end{array} \right\}$.

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

[C4] $\forall o' \in o \left\{ \begin{array}{l} (o' \in lo \to \{o'' \mid (o'', o') \in oo\} = \varnothing) \wedge \\ (o' \notin lo \to \{o'' \mid (o'', o') \in oo\} \neq \varnothing) \end{array} \right\}$.

SCQL provides for the following operations serving as not leaf ones: composition, transformation, union, and minus. Let $comp$ be set of all composition operations, $trans$ be transformation operation set, $union$ be union operation set, and $minus$ be minus operation set. All they are subset of the general operation set: $trans \subset o$, $union \subset o$, $minus \subset o$, $comp \subset o$; and they are not leaf operations:

[C5] $\forall o' \in (comp \bigcup trans \bigcup minus \bigcup union) \{o' \notin lo\}$.

A SCQL composition is a mathematical superposition operating on role concepts and not sets. A SCQL composition fulfills join-like transformation of nested operations: a) it selects all Cartesian product's instances having the same values of identical role concepts; b) it keeps one value for each role concept within each result instance. For example, if we have two nested operations with signatures (Person, Project(1), Project(2)) and (Project(2), Task), then their composition's result has the signature (Person, Project(1), Project(2), Task) as it is shown at Fig. 3. See Fig. 3. for illustration of how the result's instances are formed.

Person	Project(1)	Project(2)
3	5	6
7	8	6
19		1

Project(2)	Task
6	5
8	4
1	1

⟹

Person	Project(1)	Project(2)	Task
3	5	6	5
7	8	6	5
19		1	1

Fig. 3. SCQL Composition Example

Compositions' result signatures do not include one role concept several times while SQL join signatures can include several semantically identical columns. A composition's signature contains union of all role concepts of nested operations' signatures without duplication. For instance, the following SQL query's result 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 SCQL 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 property of SCQL composition is implicit construction of join predicates while SQL requires definition of join predicates in the explicit form. Implicit join predicates are constructed automatically as equality of identical role concepts of different nested operations. Since a SCQL composition does not have any additional parameter besides a set of nested operations, it is written as simple enumeration of nested operations in round brackets as ((Person, Phone), (Person, Skill Level, Skill Type)), and using path notation the same query is written as (Phone–(Person, Skill Level, Skill Type)). Here the associations (Person, Phone) and (Person, Skill Level, Skill Type) are composed by the concept "Person" without an explicit predicate. Note that the query's result signature is (Person, Phone, Skill Level, Skill Type) with the single concept "Person".

Also SCQL provides for outer compositions being similar to SQL outer join. By marking a nested operation with plus right after, one defines an outer composition as ((Person, Phone)+, (Person, Skill Level, Skill Type)). Here the association (Person, Phone) is outwardly added to the result of (Person, Skill Level, Skill Type). It is true that any outer composition is to have at least one nested operation not marked with plus; all plus-marked (outer) nested operations are to have common role concepts with non-outer ones; and all non-outer nested operations are to form a connected graph within each composition.

SCQL compositions provide for filtering as well. For this a special case of leaf operations named logical selection is used. A logical selection represents selection of all instances (combinations of role concept values) satisfying a certain predicate. Any logical selection is to be nested to a composition only and is described by one predicate. The predicate is to be based on role concepts of other operations nested to the same composition and not being logical selections. Superposing logical selections and other nested operations, a composition takes additional filtering effect: it is kept only those instances that satisfy predicates of all nested logical selections. Logical selections are written as logical predicates in round brackets as ((Person, Phone), (Person, Skill Level, Skill Type), (Skill Level="Experienced")) in full notation; in short notation the same query is written as (Phone–(Person, Skill Level="Experienced", Skill Type)). It reads as "select persons' skills of the experienced level and persons' phones". In both cases two associations and one logical selection are composed. Formally, let p be set of predicates, ls be logical selection set being subset of operation set: $ls \subset o$. Then each logical selection is characterized by one predicate: $lsp : ls \to p$. It is true that all logical selections are leaf operations nested to compositions:

[C6] $\forall ls' \in ls\{ls' \in lo \wedge \forall o' \in o\{(ls', o') \in oo \to o' \in comp\}\}$.

Another leaf operation is association selection. While logical selection content is calculated with a predicate, instances of association selections are to be stored in a database explicitly. All leaf operations of SCQL expressions are to be either association selections or logical selections. Each association selection is parameterized by one concept set only and is written as role concept enumeration in round brackets as (Person, Task). Role concept enumerations can be both references

to SCM associations as (Person, Phone) and closure projections as (Employee, Phone) (see above). Formally, let cs be association selection set being subset of general operation set: $cs \subset o$. Then there is $csrc \subseteq cs \times rc$ determining role concepts for each association selection.

SCQL minus and union have important differences from SQL analogues as well. SCQL permits nested operations' signatures to be nonequivalent while SQL requires alignment of both nested operations to have the same field quantity and type sequences. The only requirement for SCQL minus is intersection of role concept sets of nested operations. As a result, semantics of minus and union becomes more transparent and order independent. For instance, selection of phones of persons having no "experienced" skills is written in SQL with the minus operation 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 same query can be written in SCQL as ((Person, Phone) minus (Skill Level="Experienced", Skill Type, Person)). As you see, the second nested operation has not to be projected and correctly ordered as opposed to the above SQL query. The result signature of the minus is the signature of the first nested operation: (Person, Phone). And unions' signatures are formed as a simple union of nested operations' role concept sets. Formally, there is a restriction that any minus or union operation has exactly two nested operations:
[C7] $\forall o' \in (minus \cup union)\{|\{o'' \mid (o'', o') \in oo\}| = 2\}$.

The last SCQL operation used as not leaf one is a transformation. Signatures of all operations described above are completely calculated from nested operations' signatures. SCQL transformations cover all cases concerned with controlled modification of a single nested operation's signature by means of projection, grouping, and calculation of both aggregate and non-aggregate functions. To define a transformation, one should specify result role concepts on basis of a nested operation's role concepts perhaps using functions. SCQL provides for two alternative notations of result 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 notation, 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 fulfilled implicitly by role concepts not participated in 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. The same query in SQL 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
```

As you can see, the SQL query contains tables' names, a join criterion, and a grouping column specified unlike the SCQL query.

The last formal constraint that should be imposed on SCQL expressions reads "any transformation has only one nested operation":

[C8] $\forall o' \in trans \left\{ \left| \left\{ o'' \mid \left(o'', o' \right) \in oo \right\} \right| = 1 \right\}$.

Note that signatures of all SCQL operations represent unordered sets of role concepts. Signatures of all not leaf operations, besides transformation operations, are calculated on basis of nested operation's signatures. Only transformation operations' signatures are to be specified explicitly as $transrc \subseteq trans \times rc$.

5. Conclusion

Summing all foresaid, we conclude the following. Semantically Complete Query Language (SCQL) is a declarative conceptual language not using proper relation names and based on Semantically Complete Model (SCM). All not leaf operations of SCQL expressions, 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 result signature. Signatures of all SCQL operations are unordered, and SCQL compositions 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 languages since the last require more detailed specification of the query execution way: proper relation names, join predicates, grouping fields, and other details.

As opposed to other query languages, signatures of all SCQL queries have semantic nature as they are sets of semantic role concepts, and not sequences of abstract columns being not associated with an underlying model's concepts. In addition, SCQL does not permit creation of senseless queries unlike other languages (see examples above). Path expressions are written as role concept chains without any additional parameters. Also, SCQL has introduced conceptions of closures and contexts. Closure mechanism permits querying some indirectly associated concepts as if they are associated directly. Context mechanism based on closures permits adopting SCQL queries to users' needs without query rewriting and permits browsing SCM models by end-users in the more transparent way. As a result, SCQL is useful both for end-users and for professionals.

Now SCQL is used as foundation for SCM-based client/server technology which permits to client programs to communicate with SCM servers in terms of SCQL queries. At the moment, the SCM server is backed by any existing RDBMS system, 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 certain version or higher can be now overbuilt with SCM server to create SCM-based client/server applications [13]. The next stage of SCM-based technologies' development is creation of a data integration system providing for a general access interface as a single SCM model queried by means of SCQL. It will permit using SCQL in a distributed and heterogeneous environment in the same way as in localized one. Embedding the technology to Internet will permit for users browsing structured data just as, the way semi-structured data are browsing now by means of HTML.

References

1. Bloesch A.C., Halpin T.A. Conceptual Queries using ConQuer-II // Proceedings of ER'97: 16-th International Conference on Conceptual Modeling, 1997, pp. 113-126.
2. Embley D.W., Wu H.A., Pinkston J.S., Czejdo B. OSM-QL: a calculus-based graphical query language, Tech. Report, Dept. of Comp. Science, Brigham Young Univ., Utah, 1996.
3. Halpin T.A. Information Modeling and Relational Databases. Morgan Kaufmann, San Francisco, 2001.
4. Halpin T.A. Business Rule Verbalization // In (Doroshenko A., Halpin T., Liddle S., Mayr H. eds.) Proc. of the 3rd International Conference ISTA'2004: Information Systems Technology and its Applications, Salt Lake City, GI Lecture Notes in Informatics P-48, 2004, pp. 39-52.
5. ter Hofstede A.H.M., Proper H.A., van der Weide. Formal Definition of a Conceptual Language for the Description and Manipulation of Information Models // Information Systems, 18(7), 1993, pp. 489-523.
6. ter Hofstede A.H.M., Proper H.A., van der Weide. Query Formulation as an Information Retrieval Problem // The Computer Journal, 39, 1996, pp. 255-274.
7. ter Hofstede A.H.M., Proper H.A., van der Weide. Exploiting Fact Verbalisation in Conceptual Information Modelling // Information Systems, 22(6/7), 1997, pp. 349-385.
8. Owei V., Navathe S.B., Rhee H.-S. An abbreviated concept-based query language and its exploratory evaluation. // Journal of Systems and Software, 63(1), 2002, pp. 45-67.
9. Owei V., Navathe S. A formal basis for an abbreviated concept-based query language // Data & Knowledge Engineering, 36(2), 2001, pp. 109-151.
10. Owei V., Navathe S. Enriching the conceptual basis for query formulation through relationship semantics in databases // Information Systems, 26(6), 2001, pp. 445-475.
11. Ovchinnikov V.V. A Conceptual Modeling Technique Based on Semantically Complete Model, its Applications // In (Doroshenko A., Halpin T., Liddle S., Mayr H. eds.) Proc. of the 3rd International Conference ISTA'2004: Information Systems Technology and its Applications, Salt Lake City, GI Lecture Notes in Informatics P-48, 2004, pp. 25-38.
12. Ovchinnikov V.V. A Semantically Complete Conceptual Modeling Technique // Journal of Conceptual Modeling (www.inconcept.com/jcm), 32, 2004.
13. Ovchinnikov V.V. SCM Portal (scm.lipetsk.ru), 2005.
14. Owei V. Natural language querying of databases: an information extraction approach in the conceptual query language // International Journal of human-Computer Studies, 53(4), 2000, pp. 439-492.