Tobias Jäkel, Thomas Kühn, Hannes Voigt, Wolfgang Lehner

Towards a Role-Based Contextual Database

Erstveröffentlichung in / First published in:

ADBIS: East European Conference on Advances in Databases and Information Systems. Prague, 28. – 31.08.2016. SpringerLink, S. 89 – 103. ISBN 978-3-319-44039-2.

DOI: https://doi.org/10.1007/978-3-319-44039-2_7

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

Towards a Role-Based Contextual Database

Tobias Jäkel[1], Thomas Kühn[2], Hannes Voigt[1], and Wolfgang Lehner[1]

[1] Database Technology Group, Technische Universität Dresden, Dresden, Germany
{tobias.jaekel,hannes.voigt,wolfgang.lehner}@tu-dresden.de
[2] Software Technology Group, Technische Universität Dresden, Dresden, Germany
thomas.kuehn3@tu-dresden.de

Abstract. Traditional modeling approaches and information systems assume static entities that represent all information and attributes at once. However, due to the evolution of information systems to increasingly context-aware and self-adaptive systems, this assumption no longer holds. To cope with the required flexibility, the role concept was introduced. Although researchers have proposed several role modeling approaches, they usually neglect the contextual characteristics of roles and their representation in database management systems. Unfortunately, these systems do not rely on a conceptual model of an information system, rather they model this information by their own means leading to transformation and maintenance overhead. So far, the challenges posed by dynamic complex entities, their first class implementation, and their contextual characteristics lack detailed investigations in the area of database management systems. Hence, this paper, presents an approach that ties a conceptual role-based data model and its database implementation together, to directly represent the information modeled conceptually inside a database management system. In particular, we propose a formal database model to describe roles and their contextual information in compartments. Moreover, to provide a context-dependent role-based database interface, we extend RSQL by compartments. Finally, we introduce RSQL Result Net to preserve the contextual role semantics as well as enable users and applications to both iterate and navigate over results produced by RSQL. In sum, these means allow for a coherent design of more dynamic, complex software systems.

Keywords: Role model · Query language · Contextual database · Result net

1 Introduction

Software systems are an essential part of today's life where people and devices are connected anywhere and anytime to anyone. Additionally, new devices featuring novel technologies must be integrated into running systems without downtime. Thus, software systems have become more complex today while this trend continues. Traditional approaches, like UML or ER, fail frequently[1] when confronted

[1] For a concrete example, we refer to [18, p. 88 et sqq.].

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

with requirements of highly complex, dynamic, and context-sensitive systems. Basically, they assume static entities, although real objects evolve over time and act dynamically. From a modeling and programming perspective, these issues have been addressed by introducing the role concept [18], but most of the existing approaches neglect the contextual aspect of roles [13]. In contrast, database systems, as integral part of modern software systems, lack the notion of dynamically evolving and context-dependent data objects leading to problems during design time and run time, when the role concept is implemented in the conceptual design and programming languages. During the design phase for instance, role semantics need to be transformed into simple DBMS data model semantics, i.e., relations. This process abstracts all context-dependent information and mixes it with entity and relationship information. The run time issues are a consequence of the design time problems and the DBMS's inability to represent role semantics explicitly. A DBMS stores the data by means of its data model, which in turn provides the underlying semantics. Hence, highly specialized mapping engines are required to persist run time objects in a database and all mapping engines in the software system need to be synchronized to avoid inconsistency. This results in an increased transformation and management overhead between the applications and the DBMS. Finally, there is no external DBMS interface aware of the transformation incurred by the mapping engine. This hinders users to query and navigate their contextual data model in a coherent way.

To overcome these design time and run time issues as well as account for the often neglected context-dependent information three major goals have to be achieved. In the first place, a data model as foundation capable of representing evolving complex data objects is required. Secondly, a redesigned external DBMS interface is required enabling users and applications to query on the same semantical level as role-based programming languages. Finally, a novel result representation is needed to preserve the role-based semantics in query results. The first issue is addressed by defining the *Compartment Role Object Model* [14] based *RSQL Data Model* featuring roles and compartments for context-dependent information representation. As external database interface we propose a contextual extension to *RSQL*, a query language for role-based data. Finally, we tackle the third issue by presenting the *RSQL Result Net* that preserves the context-dependent and role-based semantics between a software system and the database.

The remainder is structured as follows: The following Sect. 2 details the running example and describes its domain. Sect. 3 introduces the context-dependent *RSQL Data Model* consisting of a type level and instance level definitions. This is followed by the description of *RSQL's query language* specifications in Sect. 4. Afterwards, the notion of our novel *RSQL Result Net* and navigational operations are detailed in Sect. 5. The related work is elaborated in Sect. 6. Finally, Sect. 7 concludes the contributions.

2 Running Example

To highlight the merits of role-based data modeling, we model a small banking application as our real world scenario, extracted from [17]. In this scenario, a

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

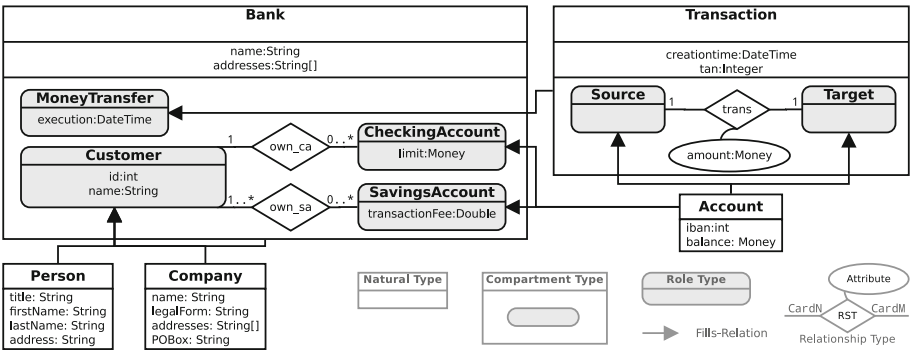

Fig. 1. Role modeling example of a small banking application

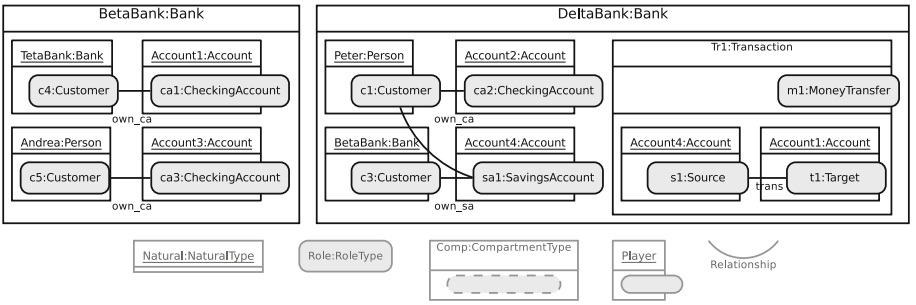

Fig. 2. Instance of the role modeling example (Fig. 1)

bank manages its customers, their accounts, as well as transactions. Customers can be persons, companies, as well as other banks. Additionally, customers may own several savings and checking accounts, and perform transactions between accounts of potentially different types. In detail, transactions embody the process of transferring money from one account to another. In addition, we specify that checking accounts must have exactly one owner, whereas savings accounts can have multiple owners. This fact is reflected by the respective cardinality constraints. Similarly, we require that one source account is linked to exactly one target account. Figure 1 depicts a possible role-based data model for this banking application. It encompasses a *Bank* as a compartment containing the roles *MoneyTransfer*, *Customer*, *CheckingAccount*, and *SavingsAccount*. The *Transaction* compartment orchestrates the money transfer between *Accounts* by means of the roles *Source*, *Target*, and the *trans* relationship constrained by one-to-one cardinality on both ends. Finally, *Persons*, *Companies*, *Banks* can play the role of a *Customer* and *Accounts* the roles *CheckingAccount*, *SavingsAccount*, *Source*, and *Target*. A simplified instance of this data model is shown in Fig. 2. It comprises two *Bank* compartment instances, **BetaBank** and **DeltaBank**. The former manages (among others) the *Customers* **TetaBank** and **Andrea** who

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

individually own a *CheckingAccount* in this bank. In contrast, the **DeltaBank** has the *Person* **Peter** as well as the former *Bank* **BetaBank** as *Customers*. Moreover, this compartment instance contains a *CheckingAccount* owned by **Peter** and a *SavingsAccount* owned by both **Peter** and the **BetaBank**. Additionally, the **DeltaBank** compartment instance contains the *Transaction* compartment **Tr1** playing the *MoneyTransfer* **m1**. Therein, **Account4** and **Account1** play the roles *Source* **s1** and *Target* **t1**, respectively, and thus, represent a transaction from **BetaBank's** savings account to **TetaBank's** checking account. Each role is placed at the border of its respective player. For brevity, we left out the individual attributes. Henceforth, the data model is used as a running example.

3 Formal Foundation

This section introduces a data model featuring the notion of compartments and context-dependent roles. In particular, this data model is strongly influenced by the *combined formal model for roles* [14] and Dynamic Typles [11,12]. Thus, a subset of the former is employed as formal foundation to extend the notion of dynamic tuples and represent compartments with context-dependent roles.

Generally, we distinguish between three meta types: Natural Types, Compartment Types, and Role Types. To discern these kinds, three ontological properties are employed, i.e., *Rigidity*, *Foundedness*, and *Identity* [5–7,16]. Both *Natural Type* and *Compartment Types* are classified as rigid with a unique identity, whereas only the latter is founded. In contrast to them, *Role Types* are not rigid [8] and founded with a derived identity. Consequently, **Person** and **Account** are considered *Natural Types*, whereas **Bank** and **Transaction** as *Compartment Types* (cf. Fig. 1). Role instances depend on the identity of their player and the existence of their context [16] (i.e., compartment). Hence, instances of a rigid type can play instances of role types. For brevity, we omit attributes and relationships from these definitions and focus on the notion of compartments.

Definition 1 (Schema). *Let NT, RT, and CT be mutual disjoint sets of Natural Types, Role Types, and Compartment Types, respectively. Then a* Schema *is a tuple $\mathcal{S} = (NT, RT, CT, \mathit{fills}, \mathit{parts})$ where $\mathit{fills} \subseteq (NT \cup CT) \times RT$ is a relation and $\mathit{parts} : CT \to 2^{RT}$ is a total function for which the following axioms hold:*

$$\forall rt \in RT \; \exists t \in (NT \cup CT) : (t, rt) \in \mathit{fills} \tag{1}$$

$$\forall ct \in CT : \mathit{parts}(ct) \neq \emptyset \tag{2}$$

$$\forall rt \in RT \; \exists! ct \in CT : rt \in \mathit{parts}(ct) \tag{3}$$

In particular, the schema definition collects the three entity kinds into their respective sets. Moreover, it defines two relations between those entity kinds. First, *fills* declares that a rigid type (either compartment or natural type) fulfills a role type, such that each role type is filled by at least one rigid type (1). Second,

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

parts collects the set of role types contained in each compartment type. In detail, it is required that there is no empty compartment type, i.e., where *parts* returns an empty set (2), and each role type is part of exactly one compartment type (3).

On the instance level natural types, role types, and compartment types are instantiated to naturals, roles, and compartments, respectively to handle context-dependent information of roles [14].

Definition 2 (Instance). *Let $\mathcal{S}$ be a schema and N, R, and C be mutual disjoint sets of Naturals, Roles, and Compartments, then an* instance *of $\mathcal{S}$ is a tuple $\mathfrak{i} = (N, R, C, type, plays)$, where $type : (N \to NT) \cup (R \to RT) \cup (C \to CT)$ is a labeling function and $plays \subseteq (N \cup C) \times C \times R$ a relation. Moreover, $O := N \cup C$ denotes the set of all objects in $\mathfrak{i}$. To be a* valid *instance of schema $\mathcal{S}$, instance $\mathfrak{i}$ must satisfy the following axioms:*

$$\forall (o, c, r) \in plays : (type(o), type(r)) \in fills \ \wedge type(r) \in parts(type(c)) \quad (4)$$

$$\forall (o, c, r), (o, c, r') \in plays : r \ \neq r' \Rightarrow type(r) \neq type(r') \quad (5)$$

$$\forall r \in R \ \exists! o \in O \ \exists! c \in C : \ (o, c, r) \in plays \quad (6)$$

In general, an instance of a schema is a collection of compartment, role, and natural instances together with their individual interrelations. In particular, the *type* function maps each instance to its type. Moreover, the *plays*-relation is the instance level equivalent of the *fills* relation and the *parts* function, as it identifies those objects (either natural or compartment) playing a role in a certain compartment. *Valid* instances are required to be consistent to a schema, i.e., they satisfy the three axioms. In detail, axiom (4) ensures the conformance of the *plays* relation to *fills* and *parts* on the type level (4). Next, axioms (5) and (6) enforce that an object can play only one role of a certain type in one compartment and that each role has exactly one player and is contained in a distinct compartment, respectively. Notably objects can still **play** multiple roles of the same type simultaneously, however these roles must be part of distinct compartments, e.g., a person can play multiple customer roles as long as they belong to different banks. This allows us to define Dynamic Tuples for complex context-dependent entities.

Definition 3 (Dynamic Tuple). *Let $\mathcal{S}$ be a schema, $\mathfrak{i}$ a valid instance of $\mathcal{S}$, and $o \in O$ is an object of type t, i.e., $type(o) = t$. A* Dynamic Tuple *$d = (o, F, P)$ is then defined with respect to the played roles and featured roles given as:*

$$F := \{\{r \mid (r, rt) \in \overline{F_o}\} \mid rt \in RT\} \ \textbf{with} \ \ \overline{F_o} := \{(r, type(r)) \mid (o, \_, r) \in plays\}$$

$$P := \{\{r \mid (r, rt) \in \overline{P_o}\} \mid rt \in RT\} \ \textbf{with} \ \ \overline{P_o} := \{(r, type(r)) \mid (\_, o, r) \in plays\}$$

In detail, a dynamic tuple is defined to capture the current rigid instance, all the roles it currently plays, and all the roles it contains. However, as an object can play and contain multiple roles of the same type, they are grouped by their type into the set F of filled roles and P of participating roles, respectively. If the set of currently filled or participating roles is empty, i.e., no role is played

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016", S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

or featured in a given object, the corresponding set is empty, denoted as $\emptyset$. In sum, this definition captures both dimensions of dynamic complex entities. Still, such entities exist in many different configurations with respect to types of the played and participating roles.

Definition 4 (Configuration). *Let $\mathcal{S}$ be a schema and $t \in NT \cup CT$ a type; then a* Configuration *of an instance of t is given as $c = (t, FT, PT)$, where $FT \subseteq \{rt \mid (ot, rt) \in \mathit{fills}\}$ and $PT \subseteq \mathit{parts}(t)$. In particular, a given dynamic tuple $d = (o, F, P)$ (with $\mathit{type}(o) = t$) in a valid instance $\mathfrak{i}$ of $\mathcal{S}$ is in exactly one* Configuration $c_o = (t, \{rt \mid (\_, rt) \in \overline{F_o}\}, \{rt \mid (\_, rt) \in \overline{P_o}\})$.

In this way, a configuration of an instance is determined by the types of roles currently played and contained. Thus, playing multiple roles of the same role type as well as containing multiple roles of the same type simultaneously does not affect the configuration. To illustrate these definitions, we discuss the following three dynamic tuples which are an expansion of instances depicted in Fig. 2:

$$d_{Account_1} := (Account_1, \{\{ca_1\}, \{t_1\}\}, \emptyset)$$
$$d_{DeltaBank} := (DeltaBank, \emptyset, \{\{c_1, c_2, c_3\}, \{sa_1, sa_2\}, \{ca_2, ca_5\}, \{m_1, m_2, m_3\}\})$$
$$d_{Tr_1} := (Tr_1, \{\{m_1\}\}, \{\{s_1\}, \{t_1\}\})$$

The first dynamic tuple represents $\mathbf{Account}_1$ that plays both a *CheckingAccount* and a *Target* role, but no participating roles, because the account is a natural instance. Consequently, its configuration is $c_1 = (Account, \{CheckingAccount, Target\}, \emptyset)$. In contrast, the *Bank* **DeltaBank** currently does not play any role, but has multiple participating roles of type *Customer*, *CheckingAccount*, *SavingsAccount* and *MoneyTransfer*. As such, $c_2 = (Bank, \emptyset, \{Customer, CheckingAccount, SavingsAccount, MoneyTransfer\})$ is its configuration. For each of these types there is a separate set of roles in F. Last but not least, the compartment $\mathbf{Tr}_1$ is playing the *MoneyTransfer* role and is featuring a *Source* and a *Target* role. In turn, its configuration is $c_3 = (Transaction, \{MoneyTransfer\}, \{Source, Target\})$. In conclusion, dynamic tuples of natural instances can only have filled roles, whereas compartment types can have both filled and participating roles.

To conclude the definition of dynamic tuples, we define both endogenous and exogenous relations. The former allows us to navigate into the filled and participating roles of a particular dynamic tuple, whereas the latter allows to navigate from one dynamic tuple to another by means of a particular role.

Definition 5 (Endogenous Relations). *Let $\mathfrak{i} = (N, R, C, type, plays)$ be a valid instance of an arbitrary schema $\mathcal{S}$, $o \in O$ an object in $\mathfrak{i}$, and $d = (o, F, P)$ the corresponding dynamic tuple. Then d* plays *a role $r \in R$ iff $(r, \_) \in \overline{F_o}$. Similarly, d* features *a role $r \in R$ iff $(r, \_) \in \overline{P_o}$.*

Basically, this lifts the notion of playing and featuring roles to the level of dynamic tuples. Consider, for instance, the dynamic tuple d_{Tr_1} currently *plays*

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

m_1 and *features* s_1 and t_1. While these relations allow to navigate within a dynamic tuple, the *Exogenous Relations* permit navigation between dynamic tuples.

Definition 6 (Exogenous Relations). *Let* $\mathfrak{i} = (N, R, C, type, plays)$ *be a valid instance of an arbitrary schema* $\mathcal{S}$*,* $o, p \in O$ *be two objects in* $\mathfrak{i}$*, and* $a = (o, F_a, P_a)$*,* $b = (p, F_b, P_b)$ *their respective dynamic tuples. Then* a *is* featured in b *with* $r \in R$*, iff* a *plays* r *and* b *features* r*. Similarly, its inverse is denoted as* b contains r *played by* a*.*

In general, *featured in* and *contains* represent the various interrelations between objects on the instance level lifted to dynamic tuples. For instance, the dynamic tuple $d_{Account_1}$ is *featured in* the transaction d_{Tr_1} (playing the role t_1). Next, the transaction d_{Tr_1} itself is *featured in* the $d_{DeltaBank}$ (playing m_1), which also *contains* the $d_{BetaBank}$. In sum, both relations are used to build our novel result set graph and provide role-based data access (see Sect. 5). In particular, endogenous relations are utilized to enable users to navigate within a dynamic tuple while exogenous relations are used to navigate from one dynamic tuple to another one.

4 RSQL Query Language

To fully support context-dependent roles, a novel query language is required capturing the previously defined notions. Thus, we introduce compartments as first-class citizen in RSQL to retain the contextual role-based semantics in the DBMS's communication interface. In detail, we discuss the syntax and semantics of RSQL's extended `SELECT` statements and how this is related to the data model's concepts defined in Sect. 3.

4.1 RSQL Syntax

RSQL consists of three language parts, the data definition language (DDL), the data manipulation language (DML), and the data query language (DQL). Based on our previous work [11,12], DDL and DML for compartments are straight forward, hence we focus on the DQL only.

The data query language consists of a `SELECT` statement, that is illustrated in Extended Backus-Naur Form (EBNF) in Fig. 3. Generally, that statement consists of three parts: (i) projection, (ii) schema selection, and (iii) an attribute filter. The first one limits the result to the specified types and attributes. The schema selection is the most complex part, specifying configurations of the desired dynamic tuples and dependencies between them. In general, the schema selection consists of a nonempty set of ⟨*config-expressions*⟩, each specifying a set of valid configurations. Those will be used in query processing to decide, whether a dynamic tuple is in a query-relevant configuration. A ⟨*config-expression*⟩ itself contains three parts: (i) the rigid type, (ii) a featuring clause describing the participating dimension of the data model, and (iii) a playing clause denoting the filling dimension. Both, the participating and filling dimension are optional in a

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

⟨*select*⟩ ::= `SELECT` ⟨*projection-clause*⟩ `FROM` ⟨*from-clause*⟩
`(WHERE` ⟨*where-clause*⟩)?

⟨*from-clause*⟩ ::= ⟨*config-expression*⟩ (`,` ⟨*config-expression*⟩)*

⟨*config-expression*⟩ ::= ⟨*rigid-name*⟩ ⟨*abbreviation*⟩
`(FEATURING` ⟨*log-expression*⟩)? (`PLAYING` ⟨*log-expression*⟩)?

⟨*log-expression*⟩ ::= ⟨*rt-def*⟩ | ⟨*log-expression*⟩ ⟨*junctor*⟩ ⟨*log-expression*⟩

⟨*rt-def*⟩ ::= (⟨*rt-name*⟩)? ⟨*rtAbbreviation*⟩

⟨*op*⟩ ::= `AND` | `OR` | `XOR`

Fig. 3. Data query language syntax

⟨*config-expression*⟩. Additionally, the featuring clause is only allowed, if the rigid is a compartment type, because natural types cannot feature role types. Finally, an optional `WHERE` clause completes the `SELECT` statement. Here, users declare the value-based filter for resulting dynamic tuples.

Example Query. The example shown in Fig. 4 is based on the schema presented in Fig. 1 and illustrates an RSQL query involving four ⟨*config-expressions*⟩. This particular query searches for bank customers of a bank and their outgoing money transfer related information from a checking account or savings account, i.e. all transactions where that particular bank customer sends money to another account. The first ⟨*config-expression*⟩ references all configurations consisting of the compartment *Bank* as rigid type and have at least the role type *Customer* in the playing clause. The second ⟨*config-expression*⟩ aims at *Accounts* that either play roles of the type *CheckingAccount* or *SavingsAccount*, and *Source*. These ⟨*config-expressions*⟩ have one dimension only, because its rigid type is a natural type. The transaction is referenced in the third ⟨*config-expression*⟩ and describes a set of configurations that has a *Transaction* as rigid type and at least one role of type *MoneyTransfer*. Additionally, the *Source* role of the *Accounts*, specified in the second ⟨*config-expression*⟩, has to participate in this compartment, which is denoted in the featuring clause by rereferencing the abbreviation of the desired role types. This ⟨*config-expression*⟩ is two-dimensional, because it describes the internal and external expansion of this particular compartment type. The last

```
SELECT * FROM Bank bc PLAYING Customer c,
          Account a PLAYING (CheckingAccount ca
                  XOR SavingsAccount sa) AND Source s,
          Transaction t FEATURING s PLAYING MoneyTransfer m,
          Bank b FEATURING m AND c AND (ca XOR sa)
```

Fig. 4. Example SELECT query

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

⟨*config-expression*⟩ describes the *Bank* compartment type that ties the roles previously described, together.

4.2 Data Model Concepts in RSQL

RSQL is a specially tailored query language for the role-based contextual data model defined in Sect. 3, thus, the data model concepts are directly represented in RSQL. In detail, RSQL leverages the two main features complex schema selection and overlapping Dynamic Tuples. The first feature is based on the idea that entities may start or stop playing several roles during runtime, and thus, change their schema dynamically. This is captured in configurations, that enable a complex object definition consisting of a rigid type and role types in two dimensions. Hence, instances of that certain type never change their type, but may vary their schema by changing the configuration. RSQL realizes this complex schema selection by a ⟨*config-expression*⟩ that defines the minimal schema a valid entity needs to have. The second feature is based on the two-dimensionality of roles which requires a role to be part of two different dynamic tuples; once in the filling dimension and once in the participating dimension. This overlapping information can be utilized in query writing to denote interrelated ⟨*config-expressions*⟩. Thus, a role type may be part of several ⟨*config-expressions*⟩ because the corresponding configurations overlap. The example query, shown in Fig. 4, exhibits several overlapping ⟨*config-expressions*⟩, for instance, the first one consisting of a compartment type *Bank bc* which has to play *Customer c* role. There, the *Customer* role type is present in the filling dimension denoted in the playing clause. Additionally, the same *Customer* role *c* is part of the *Bank b* compartment type, but in the participating dimension. Consequently, the first and fourth ⟨*config-expressions*⟩ overlap in the role type *Customer*.

5 RSQL Result Net

To preserve the role-based contextual semantics in the result, we introduce the **R**SQL Res**u**lt **N**et (RuN) enabling users to iterate over dynamic tuples and navigate along the roles to connected dynamic tuples. In particular, the navigation leverages the overlapping roles of dynamic tuples. The query result itself is an instance of the previously defined data model, hence, the query language is self-contained. Generally, RuN provides various dynamic tuples that are interconnected to each other by overlapping roles. Moreover, only queried role types are included in the result's dynamic tuples, even if the stored dynamic tuples play or feature additional roles.

RuN offers two general options to navigate in the result. Firstly, endogenous navigation path (Definition 5) to access dynamic tuple internal information. Secondly, exogenous navigation path (Definition 6) to jump from one dynamic tuple or its roles to related dynamic tuples. Each RuN is accessed by a cursor that is returned to users or applications. This cursor initially points to the first returned dynamic tuple of the first referenced ⟨*config-expression*⟩. Generally, each cursor

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

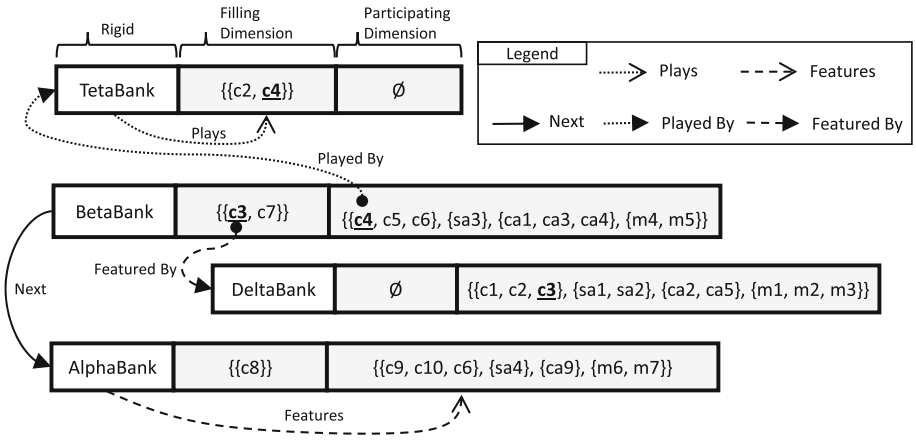

Fig. 5. Dynamic tuple navigation paths (excerpt)

provides the *Next* functionality to iterate over the set of type T, while T can be either a dynamic tuple or a role type. The *Close* functionality closes an open cursor and finalizes the iteration process on a cursor. A complex example of RuN is given in Fig. 5 illustrating endogenous as well as exogenous navigation paths. It is an extension of instance illustrated in Fig. 2 and the query shown in Fig. 4 to show all navigation paths. For the sake of clarity, we omitted redundant navigation paths in the illustration, but discuss more options in the explanation.

Endogenous Navigation. A dynamic tuple is by definition a combination of a rigid type, the set of played roles, and a set of featured roles. While iterating RSQL's result net, users want to access information about roles played by and featured in the current dynamic tuple. Functionalities providing access to this information are realized by endogenous navigation paths, in particular, by *Plays* and *Features*. Both options are based on the endogenous relation (see Definition 5).

Using the *Plays* navigation path, users are able to access a set of played roles in the filling dimension. This functionality can have two different inputs. First, a dynamic tuple only and second a dynamic tuple and a set of role types. The first one aims for accessing roles by their dynamic tuple definition, hence, the complete dimension as tuple of role sets is returned. In contrast, the second option accesses roles for a given role type and returns a new cursor to iterate over the resulting set. Therefore, this function consumes not only a dynamic tuple, but additionally a role type. Using the *Features* navigation path, users are able to access a set of featured role sets in the participating dimension. Thus, the *Features* set is created like the *Plays* set and these sets contain a set for each queried role type. By definition this path is only available for dynamic tuples having a compartment type as rigid type, because naturals cannot feature any roles. This navigation path functionality consumes either a dynamic tuple or

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

a dynamic tuple and a role type. The first input option returns the complete dimension, whereas the second only roles of the specified type. In sum, both endogenous functionalities work similar, but differ in the dimension they address.

Imagine the example RuN illustrated in Fig. 5 and a cursor pointing on the dynamic tuple **BetaBank**. Using *Plays* on this dynamic tuple by also providing the role type *Customer* would return a new cursor to iterate over the set of customer roles $\{c3, c7\}$. Utilizing the *Features* functionality on this dynamic tuples without providing a certain role type, the user will get the set $\{\{c4, c5, c6\}, \{sa3\}, \{ca1, ca3, ca4\}, \{m4, m5\}\}$. Returning the tuple instead of a set of roles gives users more flexibility in exploring roles of a dynamic tuple.

Exogenous Navigation. The exogenous navigation connects various dynamic tuples to each other by information provided by the query and the schema. RuN provides three exogenous navigation paths that are also illustrated in Fig. 5, but with solid black arrows. The first exogenous navigation path to navigate through RuN is an *iteration* implemented in the *Next* functionality that iterates over equally configured dynamic tuples. For instance, imagine the example presented in Fig. 5 and the initial cursor pointing to **BetaBank**. The *Next* functionality moves the cursor forward and gives access to the **AlphaBank** dynamic tuple.

The second exogenous navigation path is *Played By* and connects dynamic tuples that share a particular role. Here, overlapping information of dynamic tuples and the *contains* definition are leveraged to connect them. Technically, the *Played By* navigation path is used to navigate from a role that is featured in one dynamic tuple to the dynamic tuple this particular role is played in. To be connected by this path, the first dynamic tuple shares a role of its *participating* dimension with another dynamic tuple in the *filling* dimension. Thus, this functionality consumes a role and provides a dynamic tuple. Exemplarily, imagine a cursor pointing to the customer $c4$ in the participating dimension of the dynamic tuple **BetaBank** (accessing this particular role is explained in Endogenous Navigation). Executing *Played By* on this particular role will return the dynamic tuple **TetaBank**, because there customer $c4$ is in the filling dimension.

The third navigation path *Featured By* is the opposite of *Played By*. It also takes advantage of the overlapping information, but, in contrast to *Played By*, it connects dynamic tuples where the first one shares a role of its *filling* dimension with a role in the *participating* dimension of the other dynamic tuple. For this connection the *featured in* relation specified in Definition 6 is utilized. Like the *Played By* functionality, the *Featured By* consumes a role and returns the related dynamic tuple to the user. For instance, imagine the role $c3$ in the filling dimension of the dynamic tuple **BetaBank**, as illustrated in Fig. 5. A *Featured By* on this particular role aims for accessing the corresponding compartment and, thus, returns the dynamic tuple **DeltaBank**.

Complex Navigation Example. This example navigation is based on the query presented in Fig. 4 and the RSQL Result Net depicted in Fig. 5. Assume,

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

the initial cursor points to the **BetaBank** dynamic tuple. To explore the participating customer roles, the user applies the *Features* functionality by providing the role type *Customer*. This results in a cursor pointing on the customer role $c4$. Next the user searches for information about the player of this particular role, thus, uses the *Played By* functionality resulting in the dynamic tuple **TetaBank**. Additionally, the user is interested in all other played customer roles of the **TetaBank**. For this purpose, the user employs the *Plays* functionality by also providing the *Customer* role type. The new cursor points to the role $c2$. Finally, the user utilizes the *Featured By* navigation path and gets the dynamic tuple **DeltaBank** to get the information about the compartment this role $c2$ is featured in. Afterwards, the user continues with role $c5$ of the **BetaBank** by iterating to the next role in the set of played customer roles. All cursors opened to explore information related to customer role $c4$ will be closed automatically. From the $c5$ role users can repeat the procedure they used while exploring information regarding $c4$ or they go a different path[2]. After collecting all desired information of customer roles featured in **BetaBank**, the user moves on with the next dynamic tuple by applying the *Next* functionality resulting in the initial RuN cursor moving to **AlphaBank**.

6 Related Work

The concept of roles was introduced in the late 1970 s by Bachman and Daya [1]. The idea of separating the core of an object from its context-dependent and fluent parts has become popular especially in the modeling community. Steimann has surveyed various role modeling approaches until 2000 [18] and based on this research he defined 15 properties usually attached to the concept of roles. More recent approaches in modeling and programming with role-based models are detailed in [13]. Additionally, the authors extended Steimann's properties to capture context-dependent features.

In general, there are two trends in role-based and contextual data management. Firstly, developing highly specialized mapping engines that map the role semantics to traditional ones and store the data in conventional data stores. Secondly, implementing new data models into a DBMS including new query processing and data access techniques. Using specialized mapping engines simplifies storing data by abstracting the database interface. However, the data store remains the same, including the communication interface and result representation. Standard SQL queries on relational stored role-based data provide only relational results without any role-based and contextual semantics. Those semantics are vanished in the mapping process and need to be reconstructed by the mapping engine during run time. In the worst case, manual query writing becomes impossible, because the role and contextual semantics are lost and role related information is mixed with entity information. ConQuer [2], for instance, is a query language for fact-oriented models featuring weak role semantics. However, ConQuer can be seen as mapping engine, because ConQuer queries are

[2] The dynamic tuple the role $c5$ is played by, is not shown in the example.

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

transformed into standard SQL queries. The user gets the impression of relying on an Object Role Modeling [9] database, in fact the data store is a conventional relational one. Furthermore, ConQuer focuses on the query language only without considering the result representation at all. Moreover, mapping engines from role-based software to traditional data stores exist. For instance, the Role Relational Mapping [4] maps object-roles onto a relational representation for persisting and evolving runtime objects. It was designed to store, evolve, and retrieve role-based objects in a relational data store, hence, neither a query language nor a proper result representation has been developed.

The second trend is represented, for example, by the Information Networking Model (INM) [15] and DOOR [19]. The former features a data model, a query language called IQL [10], and a key-value store implementation [3]. Because the data model is hierarchically structured, they designed IQL XML-like. Furthermore, like the RSQL Result Net, IQL provides an INM instance as result. The storage layer of the INM database is an adapted key-value store utilizing different search strategies for query answering, but by design, the storage itself cannot take advantage of the semantics of the data model. Rather, they implemented a special INM layer inside of the DBMS that manages the meta information and data access [3]. Another representative of the data model implementation option can be seen in DOOR [19] designed to be an object store having role extensions to handle role-semantics. The data model utilizes special playing semantics to connect roles to their player, but lack the notion of compartments or contexts. Nevertheless, the problems of object stores like unsupported views, limited number of consistency constraints, and highly complex query optimizations remain unresolved and the external DBMS interface is undefined.

7 Conclusions

Today's highly complex and dynamic evolving software systems pose new challenges to the modeling and programming community. As consequence of the new requirements, the role concept has been established to describe dynamic entity expansion. Unfortunately, most role-based approaches neglect the context-dependent aspect of roles and do not provide a holistic view on software systems by considering databases as integral part of them. This results in transformation overhead during design and run time as well as high effort in maintenance. Within this paper, the design time issues were addressed by the RSQL Data Model which builds the foundation for direct representation of roles and compartments in a DBMS. On this basis, we proposed a RSQL query language extension to provide role-based contextual access to the database and to cope with the run time issues. Furthermore, we introduced the RSQL Result Net to preserve the contextual role semantics in results produced by RSQL query language. In particular, we examined endogenous and exogenous navigation paths in our result net to enable role-specific data access for interconnected dynamic tuples. These connections are realized by overlapping information obtained from the dynamic tuples, the schema, and the query.

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016", S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

Acknowledgments. This work is funded by the German Research Foundation (DFG) within the Research Training Group "Role-based Software Infrastructures for continuous-context-sensitive Systems" (GRK 1907).

References

1. Bachman, C.W., Daya, M.: The role concept in data models. In: International Conference on Very Large Data Bases, pp. 464–476. VLDB Endowment (1977)
2. Bloesch, A., Halpin, T.: Conquer: a conceptual query language. In: Thalheim, B. (ed.) ER 1996. LNCS, vol. 1157, pp. 121–133. Springer, Heidelberg (1996)
3. Chen, L., Yu, T.: A semantic DBMS prototype. In: Parsons, J., Chiu, D. (eds.) ER Workshops 2013. LNCS, vol. 8697, pp. 257–266. Springer, Heidelberg (2014)
4. Götz, S., Richly, S., Aßmann, U.: Role-based object-relational co-evolution. In: Proceedings of 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE 2011) (2011)
5. Guarino, N., Carrara, M., Giaretta, P.: An ontology of meta-level categories. In: Principles of Knowledge Representation and Reasoning: Proceedings of the Fourth International Conference, pp. 270–280. Morgan Kaufmann (1994)
6. Guarino, N., Welty, C.A.: An overview of OntoClean. In: Staab, S., Studer, R. (eds.) Handbook on Ontologies, pp. 201–220. Springer, Heidelberg (2009)
7. Guizzardi, G.: Ontological foundations for structure conceptual models. Ph.D. thesis, Centre for Telematics and Information Technology, Enschede, Netherlands (2005)
8. Guizzardi, G., Wagner, G.: Conceptual simulation modeling with onto-UML. In: Proceedings of the Winter Simulation Conference, WSC 2012, pp. 5:1–5:15. Winter Simulation Conference (2012)
9. Halpin, T.: ORM/NIAM object-role modeling. In: Handbook on Architectures of Information Systems (1998)
10. Hu, J., Fu, Q., Liu, M.: Query processing in INM database system. In: Chen, L., Tang, C., Yang, J., Gao, Y. (eds.) WAIM 2010. LNCS, vol. 6184, pp. 525–536. Springer, Heidelberg (2010)
11. Jäkel, T., Kühn, T., Hinkel, S., Voigt, H., Lehner, W.: Relationships for dynamic data types in RSQL. In: Datenbanksysteme für Business, Technologie und Web (BTW) (2015)
12. Jäkel, T., Kühn, T., Voigt, H., Lehner, W.: RSQL - a query language for dynamic data types. In: Proceedings of the 18th International Database Engineering & Applications Symposium, pp. 185–194 (2014)
13. Kühn, T., Leuthäuser, M., Götz, S., Seidl, C., Aßmann, U.: A metamodel family for role-based modeling and programming languages. In: Combemale, B., Pearce, D.J., Barais, O., Vinju, J.J. (eds.) SLE 2014. LNCS, vol. 8706, pp. 141–160. Springer, Heidelberg (2014)
14. Kühn, T., Stephan, B., Götz, S., Seidl, C., Aßmann, U.: A combined formal model for relational context-dependent roles. In: International Conference on Software Language Engineering, pp. 113–124. ACM (2015)
15. Liu, M., Hu, J.: Information networking model. In: Laender, A.H.F., Castano, S., Dayal, U., Casati, F., de Oliveira, J.P.M. (eds.) ER 2009. LNCS, vol. 5829, pp. 131–144. Springer, Heidelberg (2009)
16. Mizoguchi, R., Kozaki, K., Kitamura, Y.: Ontological analyses of roles. In: 2012 Federated Conference on Computer Science and Information Systems (FedCSIS), pp. 489–496. IEEE (2012)

Final edited form was published in "Advances in Databases and Information Systems. Prague 2016",
S. 89 - 103. ISBN: 978-3-319-44039-2
https://doi.org/10.1007/978-3-319-44039-2_7

17. Reenskaug, T., Coplien, J.O.: The DCI architecture: a new vision of object-oriented programming. An article starting a new blog: (14pp) (2009). http://www.artima.com/articles/dci_vision.html
18. Steimann, F.: On the representation of roles in object-oriented and conceptual modelling. Data Knowl. Eng. **35**(1), 83–106 (2000)
19. Wong, R., Chau, H., Lochovsky, F.: A data model and semantics of objects with dynamic roles. In: 13th International Conference on Data Engineering, April 1997, pp. 402–411. IEEE (1997)