2012 38th Euromicro Conference on Software Engineering and Advanced Applications

ORCA: Architecture for Business Tier Components Driven by Dynamic Adaptation and Based on Call Level Interfaces

Óscar Mortágua Pereira[(1)], Rui Luís Aguiar[(2)]
Instituto de Telecomunicações
DETI, University of Aveiro
3810-193 Aveiro, Portugal
omp@ua.pt[(1)], ruilaa@ua.pt[(2)]

Maribel Yasmina Santos
Centro Algoritmi
University of Minho
4800-058 Guimarães, Portugal
maribel@dsi.uminho.pt

***Abstract*— Call Level Interfaces (CLI) play a key role in database applications whenever performance is a key requirement. SQL statements are encoded inside strings this way keeping the power and the expressiveness of the SQL language. Unfortunately, despite this significant advantage, CLI do not promote the development of business tier components, much less for business tier components driven by dynamic adaptation. To tackle this CLI drawback, and simultaneously keep their advantages, this paper proposes an architecture, herein referred to as the Object-to-Relational Component Architecture (ORCA), relying on CLI for building adaptable business tiers components. ORCA has the capacity of being dynamically adapted to manage any set of SQL statements deployed at runtime. The focus of this paper is threefold: 1) present the ORCA, 2) present a proof of concept based on Java and, finally, 3) assess its performance against a standard CLI.**

Keywords- components; adaptive systems; software architecture; performance; call level interfaces; databases.

I. INTRODUCTION

Call Level Interfaces (CLI) are effective solutions for building business tiers whenever performance is a key requirement [1]. There are two main reasons: 1) SQL statements are encoded inside strings, keeping the power and the expressiveness of the SQL language and 2) CLI are low level API inducing very low overhead. In spite of these important advantages, CLI also convey some drawbacks: business tier components are not easily built from CLI, much less for business tiers driven by dynamic adaption.

Problem definition: Components have their fundamentals in component-based development which is a key topic in software engineering [2, 3]. Component-based development aims to compose software artifacts from other pre-built software artifacts [2]. At the end, a final system is not built as a unique block but as a composite of software artifacts known as components [4]. A key aspect for the success of any component is its capability of being reused and adapted [5]. Thus, a component perspective for business tiers must address two key aspects: 1) enforcement of a clear separation between the development process of business tiers from the development process of application tiers, and 2) promote a swift adaptation process of business tier components to new business needs (these new needs are mostly and mainly felt at development time of application tiers but may also happen after their deployment). CLI are the opposite of this.

1) Regarding the component perspective, programmers are easily pushed to mix source code of business tiers with source code of application tiers. Figure 1 presents a typical and simple case using JDBC [6] as an example of a CLI. This example depicts a method aimed at updating the attribute *value* of a list of orders kept in a table named *Orders*. The list is included in *List<Integer> order* and the new values are included in *List<Float> value* (see arguments of method *updOrders*). The query is prepared (selects all orders) and is executed (line 84-86); the returned relation is iterated (line 87); it is checked if the current order needs to be updated (line 88-90); the new value is retrieved (line 91) and then the old value is updated (line 92-93). Programmers play the business tier developer role when they write SQL statements and source code to execute them (line 84,85,86,87,88,92,93) and play the application tier developer role when they use the application data and retrieved data (line 88,89,90,91,92,93). In reality, CLI were not devised to avoid this tangling of roles which clearly inhibits the development process of business tier components.

2) Regarding the adaptation process, CLI were not geared to address it. All source-code needs to be manually written before being used, as shown in Figure 1. There is no way to dynamically adapt business tiers to new

```
80  void updOrders(List<Integer> order,
81                 List<Float> value)
82                              throws SQLException {
83      // some code
84      sql="Select * from Orders;";
85      st=conn.createStatement();
86      rs=st.executeQuery(sql);
87      while (rs.next() ) {
88          orderId=rs.getInt("orderId");
89          int idx=order.indexOf(orderId);
90          if ( idx!=-1) {
91              newValue=value.get(idx);
92              rs.updateFloat("value", newValue);
93              rs.updateRow();
```

Figure 1. Tangling of roles with JDBC.

978-0-7695-4790-9/12 $26.00 © 2012 IEEE
DOI 10.1109/SEAA.2012.22

business needs. Moreover, each SQL statement is managed by a block of source-code very often comprising many repeated lines of code previously written for other SQL statements. This situation is increasingly critical when the number of SQL statements increases eventually becoming difficult to be manageable in database applications with several hundreds of SQL statements.

Solution: to tackle these CLI drawbacks, a research has been carried out, in the context of Component-Based Software Engineering [2, 3]. The result of the research is a architecture for adaptable components, herein known as Object-to-Relational Component Architecture (ORCA), aimed at building business tiers. ORCA main features are the following: 1) it is based on CLI to make use of and keep their key advantages; 2) it promotes the development of components to build business tiers; 3) components are dynamically, at runtime, adapted to address new business needs.

Contribution: the main contribution of this research is threefold: 1) to present the ORCA aimed at overcoming and tackling CLI drawbacks, 2) to present a proof of concept based on JDBC and, finally, 3) to present the results of a performance assessment.

This paper is structured as follows: section II introduces CLI; section III presents ORCA; section IV presents the performance assessment; section V presents the related work and section VI presents the final conclusion.

II. CALL LEVEL INTERFACES

CLI are considered important options for building business tiers whenever performance is considered a key requirement [1]. CLI provide mechanisms to encode SQL statements inside strings, easily incorporating the power and the full expressiveness of SQL. JDBC [6] and ODBC [7] are two representatives of CLI. SQL statements are executed against the host database and the possible results they produce (only for Select statements) are locally managed by local memory structures (LMS) – (ResultSet [8] for JDBC, RecordSet [9] for ODBC). Hereafter, SQL statements will be restricted to Select, Insert, Update and Delete statements. Only CLI services directly related to the execution of SQL statements will be considered. Services such as those for managing connections to host databases are not addressed in this paper.

Key services of CLI are organized in three main categories: execution target, scrollability and updatability. Figure 2 will be referred during the next explanations.

Execution target: it comprises services related to the execution of SQL statements. They are executed as compiled-on-the-fly or pre-compiled (when they are to be reused or when they have parameters defined at runtime time (line 42)). Additionally, CLI deal differently with Select statements from the other three types of SQL statements. Select statements instantiate an LMS (line 47 - rs:ResultSet), while the other types do not, generating a value indicating the number of affected rows in the database.

Scrollabilty: it comprises services related to the scrolling process on LMS. There are two mutual-exclusive possibilities: *forward-only* (line 44) – in this case it is only possible to move forward one row at a time, (line 48); *scrollable* – in this case it is possible to move in any direction and jump several rows at a time.

Updatability: it comprises services organized in protocols to interact with data contained in LMS. There are two mutual-exclusive possibilities: *read*-only – the contents of the LMS is read-only and no changes are allowed; *updatable* (line 45) – changes may be performed on LMS (insert new rows (line 53-55), update rows (line 51-52) and delete rows (line 50). JDBC commits these changes into the host database.

```
42      sql="Select * from Person p Where p.id=?";
43      ps=conn.prepareStatement(sql,
44              ResultSet.TYPE_FORWARD_ONLY,
45              ResultSet.CONCUR_UPDATABLE);
46      ps.setInt(1,id);
47      rs=ps.executeQuery();
48      if (rs.next() ) {
49        grade=rs.getFloat("fName");
50        rs.deleteRow();
51        rs.updateFloat("grade",value);
52        rs.updateRow();
53        rs.moveToInsertRow();
54        rs.updateString("fName",fName);
55        rs.insertRow();
```

Figure 2. Typical JDBC usage.

III. ORCA

Before ORCA presentation some of its key concepts are introduced in the following order: sibling SQL statement, Business Entity, Object-to-Relational Model, Business Schema and Business Engine. Then, ORCA is presented and followed by a proof of concept.

A. Sibling SQL statements

SQL statements may be characterized by a schema, herein known as SQL schema, which comprises their type (Select or Update, Insert and Delete), their runtime parameters and their returned relations (only for Select statements). SQL statements sharing the same SQL schema, as the ones shown in Listing 1, are herein known as sibling SQL statements. The key issue is that sibling SQL statements may be managed by the same source-code leveraging this way an important aspect of components:

```
-- a simple statement
Select p.id,p.fName,p.lName
  From pilot p
-- a more complex statement
select p.id,p.fName,p.lName
 From pilot p,circuit c,classif f
 Where p.id=f.id and f.date=c.date
```

Listing 1. Sibling SQL statements.

the reuse of computation [10].

B. Business Entity

Business Entities (classes) are one of ORCA key entities. Each one is responsible for managing the execution of sibling SQL statements on behalf of application tiers. Each Business Entity implements one interface, herein known as Business Interface, which defines the needed services to deal with one SQL schema.

Select statements return relations while the remaining SQL statements do not. This leads to the need of thoroughly assessing CLI to organize their services in homogeneous and disjoint groups of services, herein known as Service Interfaces. Ten groups were identified: IExecute (IExecutePC, IExecuteFC), IResult, ISet, IRead, IForwardOnly, IScrollable, IWrite, IUpdate, IInsert and IDelete, see Figure 3.

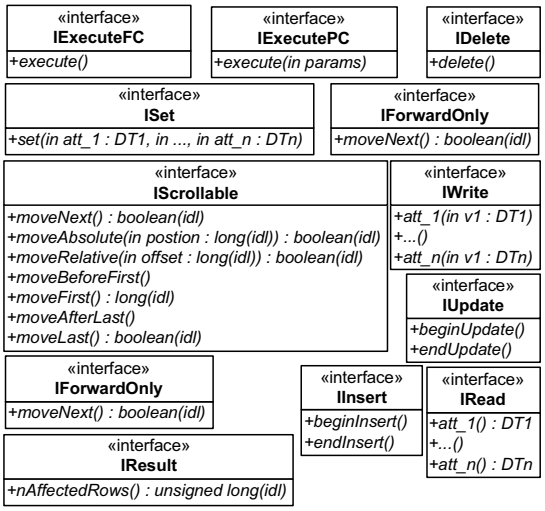

Figure 3. Service Interfaces.

1) IExecute is used for executing SQL statements. There are two facets: one for the compiled-on-the-fly statements (*IExecuteFC*) and one for the pre-compiled statements (*IExecutePC*). The argument *params* is only necessary if there are parameters defined at runtime.

2) IResult comprises one method to return the number of affected rows after the execution of any Update, Insert or Delete SQL statement.

3) ISet is used with pre-compiled Insert and Update SQL statements when one or more values for their list attributes are defined at runtime.

4) IForwardOnly is used with forward-only LMS and comprises one method to scroll one row forward.

5) IScrollable is used with scrollable LMS.

6) IRead is used to read attributes from LMS. The signature of each method is driven by the schema of the correspondent attribute. *DTn* is the correspondent SQL data type in the host programming language.

7) IWrite is used to update and insert attributes of updatable LMS. The signature of each method is driven by the schema of the correspondent attribute.

8) IUpdate is used to start and to end the execution of the update protocol on updatable LMS.

9) IInsert is used to start and end the execution of the insert protocol on updatable LMS.

10) IDelete is used to delete rows from updatable LMS.

Service Interfaces are used and aggregated to build Business Interfaces. IExecuteFC, IResult, IForwardOnly, IScrollable, IUpdate, IInsert and IDelete are not dependent on SQL schemas of SQL statements and, whenever necessary, may be shared by all Business Interfaces. IExecutePC, IRead, IWrite and ISet are dependent on SQL schemas and, therefore, need to be customized to be used in Business Interfaces.

C. Object-to-Relational Model

Object-to-Relational Model (ORM) is a model aimed at mapping schemas of SQL statements into the object-oriented paradigm. From a functional perspective, SQL statements and LMS may be organized in two groups: execute statements (Insert, Update and Delete) and select statements (Select). Select statements may be forward-only or scrollable or, read-only or updatable, as previously explained. This organization leads the ORM to be structured around three main facets, each one addressing specific features: ORM_IUD for Insert, Update and Delete statements, see Figure 4, ORM_Sro for Select statements that create read-only LMS, see Figure 5, and, finally, ORM_Sup for Select statements that create updatable LMS, see Figure 6. From these class diagrams, it is possible to foresee that ORM has the capacity of being

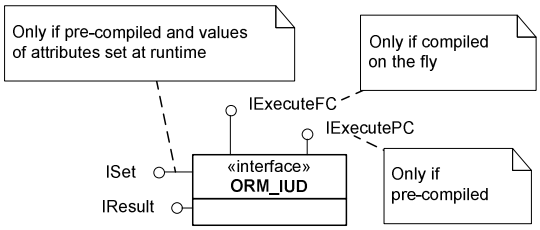

Figure 4. ORM_IUD class diagram.

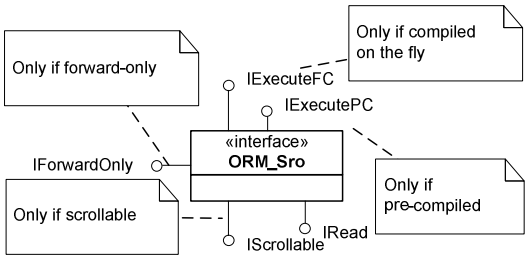

Figure 5. ORM_Sro class diagram.

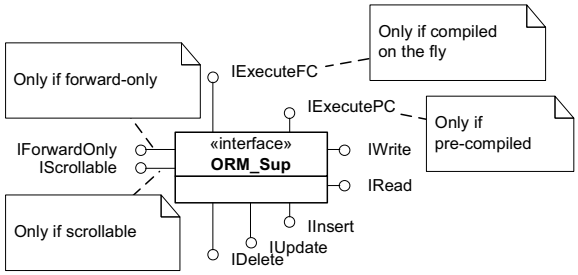

Figure 6. ORM_Sup class diagram.

adapted to specific needs of each SQL statement. For example, let's consider a Select statement that is compiled-on-the-fly and generates updatable and forward-only LMS (modeled by the ORM_Sup). The correspondent Business Interface comprises the following Service Interfaces: IExecuteFC, IFowardOnly, IRead, IWrite, IInsert, IUpdate and IDelete.

D. Business Schema

Business Schemas are entities derived from the ORM and are used to build Business Interfaces. They aggregate all the required information to build Service Interfaces from which Business Interfaces are built in accordance with the ORM. Due to the formalization process of Service Interfaces, which is based on programming interfaces, an attempt was made to formalize Business Schemas also as programming interfaces. The approach proved to have the following key advantages over other approaches, such as XML: 1) programmers wouldn't need to change its programming language; 2) programming interfaces are widely used and are an unavoidable concept in object-programming languages; 3) programming interfaces are easy to create and maintain; 4) due to the same formalization process, Business Schemas are directly inferred and defined from the Service Interfaces. Only IExecutePC, IRead and ISet need to be explicitly created for each Business Interface. They are dependent on schemas of SQL statements while the others are not. IWrite is also dependent but the methods are automatically inferred from IRead interface.

E. Business Engine

Business Engine is a component aimed at writing the source code for Business Entities and Business Interfaces from Business Schemas. The functionalities to be implemented in each Business Entity are inferred by the Business Engine, following the next rules: 1) if the Service Interface IExecuteFC is present, than the SQL statements are complied-on-the-fly; 2) if the Service Interface IExecutePC is present, then the SQL statements are pre-compiled; 3) if at least one of the following Service Interfaces IForwardOnly, IScroll, IRead, IUpdate, IInsert or IDelete is present, then an LMS must be instantiated; 4) if the Service Interface IForwardOnly is present then the LMS will be instantiated as forward only; 5) if the Service Interface IScrollable is present then the LMS will be instantiated as scrollable; 6) if at least one of the following Services Interfaces IUpdate, IInsert or IDelete is present, then the LMS is instantiated as updatable, otherwise, it will be instantiated as read-only.

F. ORCA presentation

Key entities of ORCA have just been presented. They need to be organized to promote the building process of business tier components driven by dynamic adaptation. Previous entities were organized and some additional entities were added as shown in Figure 7. There are two additional interfaces (IConfig, ISession) and an additional entity, named as *Manager*, to implement both interfaces.

IConfig interface, see Figure 8, is used to configure components derived from ORCA. The 6 main types of services are: *dbServer* to define the parameters for connections to database servers; *repository* to define the physical location of Business Entities and Business Interfaces; *addBusinessEntity* to start the Business Engine (Figure 7, 1-1) to create new Business Entities (Figure 7, 1-2); r*emoveBusinessEntity* to remove Business Entities; a*ttachStatement* to deploy and delegate the management of SQL statements to Business Entities and *dettachStatement* to undo the previous operation.

ISession interface, see Figure 9, owns a private connection to the host database and provides two methods. One generic method *createBusinessWorker* (Figure 7, 2-1) used to create instances of Business Entities (BE), herein known as Business Workers (BW) (see Figure 7, 2-2). This generic method is a key aspect in the ORCA. With a single method, and using reflection, it is possible to implement a type safe instantiation process for all Business Entities. The returned *Interface* implements one of the ORM facets previously presented: ORM_IUD, ORM_Sro or ORM_Sup. In order to keep control on the SQL statements being executed by each Business to

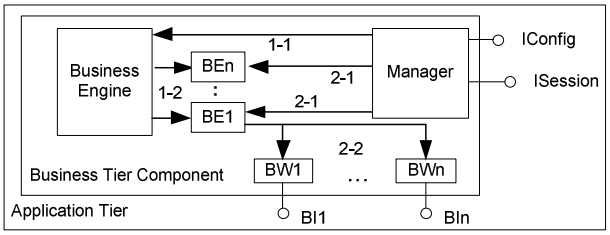

Figure 7. Block diagram of ORCA.

«interface» **IConfig**
+dbServer(in dbms : DBMS, in url : string(idl), in database : string(idl), in …) *+repository(in path : string(idl), in clear : boolean(idl))* *+addBusinessEntity(in businessSchema : Interface)* *+removeBusinessEntity(in fileName : Interface)* *+attachStatement(in stId : long(idl), in st : string(idl), in bi : Interface)* *+dettachStatement(in stId : long(idl), in bi : Interface)*

Figure 8. IConfig interface.

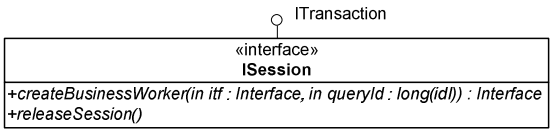

Figure 9. ISession interface.

Entity, it is not allowed to request a Business Entity manage a statement that has not been previously attached to it. The second method is used to release the open session. ISession extends *ITransaction* to manage database transactions.

Programmers of application tiers use ORCA based on an approach with four phases. First, they import a basic component derived from ORCA (not adapted yet) into the development environment. Second, they adapt the component to their specific needs by editing and sending Business Schemas to IConfig at runtime. Third, they write source code to use the component just adapted. Forth, SQL statements are deployed at runtime. Whenever necessary, the process may be repeated from the second phase to update the components being used.

G. *Proof of Concept*

A proof of concept based on Java and JDBC (sqljdbc4) for SQL Server 2008 database was developed. Tests have been concluded and a free trial may be downloaded[1].

Figure 10 presents a simplified class diagram for the component derived from ORCA: 1) ORCA is instantiated through the static method *getInstance*; 2) the configuration

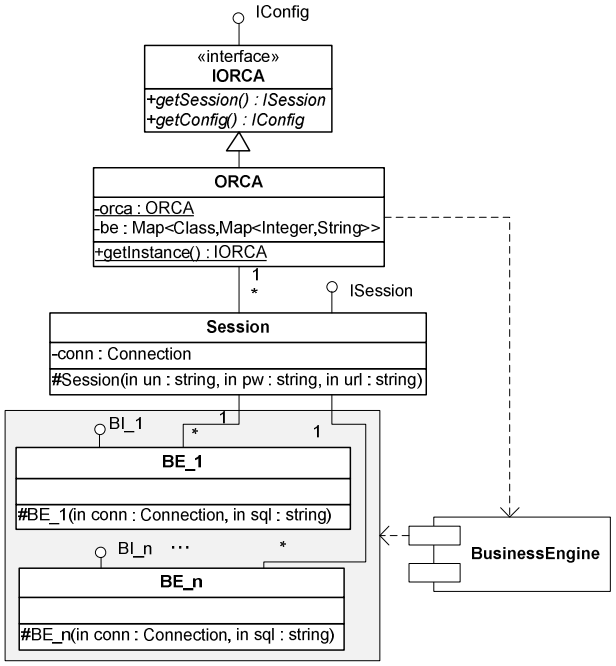

Figure 10. ORCA simplified class diagram.

[1] http://dl.dropbox.com/u/71192544/Work/Confers/SEAA/SEAA2012/ORCA.7z

is processed through IConfig; 3) ORCA keeps track of Business Entities and their associated SQL statements through the *Map* [11] *be* object; 4) Business Engine is used to create Business Entities each one able to manage sibling SQL statements deployed at runtime; 5) Business Workers are created from ISession interface and accept at instantiation time an SQL statement and a connection object to the host database.

A scenario was created to present ORCA from its usage point of view. The scenario comprises: 1) two sibling Select SQL statements: *Select * from student where id=?* (identified as s1) and *Select * from student where courseId=?* (identified as s2) to be managed by one business interface modeled by ORM_Sup and herein known as IStudent_get; 2) Insert SQL statement: *insert into student values (?,?,?,?,?,?,?)* (identified as s3) to be managed by a business interface modeled by ORM_IUD, herein known as IStudent_insert. All SQL statements are pre-compiled. Figure 11, Figure 12 and Figure 13 present details about how to use ORCA. Figure 11 shows the source code to deploy the two Business Schemas to the Business Engine. Figure 12 shows the source code to deploy and attach the SQL statements to Business Entities at runtime. SQL statements s1 and s2 are managed by IStudent_get and their ids are 1 and 2, respectively. SQL statement s3 is managed by IStudent_insert and its id is 1. Figure 13 presents a typical case for the use of a component derived from ORCA. A is

```
106  void buildBusinesEntities(IConfig c) {
107      c.addBusinessEntity(IStudent_get.class);
108      c.addBusinessEntity(IStudent_insert.class);
109  }
```

Figure 11. Configuration of Business Entities

```
130  void config(IConfig c) {
131      s1="select * from student where id=?;";
132      s2="select * from student where courseId=?;";
133      s3="insert into student values(?,?,?,?,?,?,?);";
134      c.attachQuery(1,s1,IStudent_get.class);
135      c.attachQuery(2,s2,IStudent_get.class);
136      c.attachQuery(1,s3,IStudent_insert.class);
137  }
```

Figure 12. Configuration of SQL statements.

```
69  private IORCA o;
70  //... code
71  void getStudent(int id, String name)
72                              throws SQLException,
73                                     ORCA_Exception {
74      ISession s=o.getSession();
75      IStudent_get st=
76        s.createBusinessWorker(IStudent_get.class,1);
77      st.executePC(id);
78      if (st.moveNext()) {
79          st.beginUpdate();
80          st.fName(name);
81          st.endUpdate();
82          // .. moore code
83      }
84  }
```

Figure 13.Use case of ORCA.

new session is created (line 74); a Business Worker created (line 75-76) from Student_get to manage SQL statement s1; SQL statement is executed (line 77); if a student has been found (line 78) its first name is updated (line 79-81).

From this scenario, we have shown that ORCA is suited for building business tier components driven by dynamic adaptation. Components exist as independent software units and are adaptable because: a) Business Entities are automatically built at runtime to address specific needs; 2) SQL statements are deployed at runtime to address specific users' needs; 3) each Business Entity is able to manage any SQL statement whose schema is aligned with its Business Interface. The adaptation process comprises three main activities: editing process of SQL statements, editing process of Business Schemas and the configuration process. The first activity is also required when tiers are built from CLI, not leading to any additional effort. The other two activities are specific to ORCA. In this paper we have presented a manual approach for their implementation. While the editing process of Business Schemas very probably will be a manual process, the configuration process is easily automated. Thus, the adaptation process is confined to the editing process of Business Schemas which, in our implementation, are standard Java interfaces.

IV. PERFORMANCE ASSESSMENT

The performance assessment is focused on evaluating and comparing the performance of solutions based on components derived from ORCA and solutions based on standard CLI. Java, JDBC and SQL Server 2008 have been chosen as the basic core technologies to support the assessment and the component is herein known as JORCA. The test-bed relies on a Dell Latitude E5500 Laptop, Intel Duo Core P8600 @2.40GHz, with 4.00 GB RAM, with Windows Vista Enterprise Service Pack 2 (32bits), Java SE 7, JDBC(sqljdbc4) and SQL Server 2008. The minimum system interval counter is 428ns. In order to promote an ideal environment, the following actions were taken: 1) the running threads were given the highest priority; 2) all non-essential processes/services were cancelled; 3) transactions were not used and auto-Commit (JDBC) has been always enabled; 4) a new database was created, containing one table named *target* with a single attribute named *id* of type *int* and *not null*; 5) some default SQL Server database properties were changed such as Auto Update Statistics = false and Recovery Model = Simple. The strategy followed to collect the needed measurements was based on measuring how long some code takes to execute. To achieve this goal, the method *system.nanoTime()* was used. In spite of being a very easy methodology to collect measurements, it cannot be directly applied to situations where the collected measurements are in the same order of magnitude as the time to process an empty block of code. Thus, the first step is to evaluate the impact of the act of measuring. The collected values showed that the impact is 1,284ns±428ns.

Table I. General algorithm to collect measurements.

1 repeat: 20 rounds
1.2 get a new container to keep the collected times
1.3 repeat: 250 cycles
1.3.1 start timer
1.3.2 run scripts (must take at least 171,200ns)
1.3.3 stop timer
1.3.4 keep elapsed time if it is one of the 5 best in this cycle
1.3.5 release all unnecessary objects
1.3.5 activate garbage collector
1.3.5 sleep 100ms (other system processes may need to run)
2 compute the average time of each round
3 keep the best average time

Thus, for statistic effects, the worst case was considered:1,284ns+428ns=1,712ns. From this value, and in order to keep errors below 1%, all measurements associated with the performance assessment were collected with a minimum time span of 171,200ns. In several situations it was necessary to repeat the same code as often as necessary to get a minimum of 171,200ns. To avoid additional errors with the repeating process, the code was sequentially repeated and not iteratively repeated. The general strategy followed to collect and compute each presented measurement is presented in Table I. All measurements are presented in nanoseconds (ns).

A. ORCA Overhead

Regarding performance, ORCA may be split into two main phases: 1) the creation phase which is focused on the creation of Business Entities and 2) the execution phase, which is focused on the execution of statements. The creation phase is related to activities that have no equivalent on standard CLI and, thus, the % induced overhead is 100%. The creation phase comprises: the configuration process, the creation of source code for Business Entities, their compilation, their loading into memory and, finally, their instantiation. The execution phase comprises the activities shared by standard CLI and by JORCA, which are directly related to the execution of SQL statements and, therefore, the assessment is focused on measuring the overhead induced by Business Workers. Two main approaches may be followed to carry out the performance assessment for the execution phase: 1) use JORCA and assess it against the standard JDBC API based on case studies; 2) develop a general environment aimed at evaluating the overhead induced by the wrapping process implemented by each method of each Service Interface. After some initial measurements, it came clear that the latter approach would bring significant advantages over the former approach, in several dimensions, such as: a) methods of Service Interfaces are general and not tied to any particular use case and b) an assessment based on the wrapping process may lead to a mathematical model to evaluate any scenario.

Creation phase: The collected measurements were obtained using a case study based on a scrollable ORM_Sup implementing all Service Interfaces. The collected measurements were 41,956,211ns for TP (time to create source code, compile and load Business Entities)

and 1,510ns for TI (time to instantiate a Business Worker). While TP behaves as a one-time overhead, TI is an overhead for each instantiated Business Worker. To avoid this overhead, Business Workers may be reused. Sessions were not considered in this performance assessment because performance is mostly influenced by the policy used to manage pool of connections to the host database.

Execution phase: activities related to the execution phase are basically the invocation of Service Interfaces methods. Each Service Interface method wraps a block of code of the standard CLI. Thus, the overhead may be measured by evaluating the time to execute the additional code when using a Service Interface method. To achieve this, we introduce the concept of reduced method signature (RMS). RMS derives from the widespread concept of method signature but it does not include the method name. All methods of Service Interfaces may be classified in two different groups: methods with a fixed RMS and methods with a variable RMS. Methods with fixed RMS are, by far, the major group. The only methods which do not have a fixed RMS are: *execute* with parameters and *set*. In order to predict the overhead induced by every wrapping method, it was decided to measure the finest grain overhead induced by each possible variation in RMS. Two examples: measure the induced overhead by each additional argument of any data type and measure the induced overhead by returning any data type. To achieve this goal, two types of measurements were collected as shown in Table II. TR_i are the collected measurements for methods with no arguments and returning the data types shown in the first column. Examples: *void m() {}* and *int m(){ return 1;}*. TA_i are the collected measurements for each additional argument of type *Data type*. TA_i measurements were collected using methods with 10 arguments of the same data type and returning *void*. Then, TA_i=(collectedMeasurement-3)/10 where 3 is the time to call the method *void m() {}*. This approach was validated by carrying out some additional tests using less than 10 and more than 10 arguments. From Table II it is easy to compute the overhead induced by any RMS and, therefore, of each method of each Service Interface. In spite of being important data, they do not give any insight about their impact on real cases. Thus, 4 main scenarios based on JDBC were defined: Select (S_s), Update (S_u), Insert (S_i) and Delete (S_d) scenarios to assess the execution of Select, Update, Insert and Delete statements, respectively. S_s is based on the execution of a compiled-on-the-fly Select statement (*Select * from target*). S_s comprises all types of LMS, regarding their scrollability and updatability, and measures the most usual operations: the execution of a Select statement, scrolling on the LMS, reading attributes, inserting rows, updating rows and deleting rows.

Table III presents the collected measurements and the computed overheads (the description of each column may be found at the bottom).

Table IV describes the algorithms used in each task. To avoid any SQL Server optimization process, the used table was dropped and created in each cycle (see Table IV). From Table III it is clear that the induced overhead is only noticeable on partial tasks, such as reading attributes, scrolling and writing attributes without committing them. The % overhead, for these tasks, range from 1.8% till 3.3%. In a more realistic approach, the overhead should be computed comprising a whole cycle (read protocol), such as the instantiation of a Business Worker (optional because it may be reused), the statement execution, scrolling to the first row and then read it (E+S+R). In this case the total time for a FR (best score) is 352,873+215+180=353,268ns. The overhead is TI+V (when using new Business Workers) or V (when reusing Business Workers), where

Table II. Collected measures for typical RMS in ns.

Data type	TR_i	TA_i	i	Data type	TR_i	TA_i	i
void	3	0.0	1	String	4	1.3	6
byte	6	1.1	2	float	6	2.2	7
short	6	1.7	3	double	6	2.3	8
int	6	1.3	4	boolean	6	1.1	9
long	6	2.3	5	char	6	1.6	10

Table III. Collected measurements for scenario S_s.

Rs	Task	JDBC (ns)	ORCA Overhead		%	ORCA methods (OM)
			T	ΔT(ns)		
FR	E	352,873	TR_1	3	δ	execute
FU	E	3,715,784	TR_1	3	δ	
SR	E	3,737,613	TR_1	3	δ	
SU	E	3,759,700	TR_1	3	δ	
FR	S	215	TR_9	6	2.8	moveNext
FU	S	320	TR_9	6	1.9	
SR	S	335	TR_9	6	1.8	
SU	S	335	TR_9	6	1.8	
FR	R	180	TR_4	6	3.3	IRead
FU	R	261	TR_4	6	2.3	
SR	R	261	TR_4	6	2.3	
SU	R	261	TR_4	6	2.3	
FU	CN	479	TR_1+TA_4	7.3	1.5	IWrite
SU	CN	479	TR_1+TA_4	7.3	1.5	
FU	UN	263	TR_1+TA_4	7.3	2.8	
SU	UN	263	TR_1+TA_4	7.3	2.8	
FU	CC	256,821	3*TR_1+TA_4	10.3	δ	IWrite, beginInsert,endInsert
SU	CC	263,893	3*TR_1+TA_4	10.3	δ	
FU	UC	288,890	3*TR_1+TA_4	10.3	δ	IWrite,beginUpdate, endUpdate
SU	UC	294,369	3*TR_1+TA_4	10.3	δ	
FU	D	319,435	TR_1	3	δ	Delete
SU	D	328,115	TR_1	3	δ	

RS	Type of LMS: acronym formed by a first letter F or S (forward-only or scrollable) and a second letter R or U (read-only or updatable). Ex: FR is forward-only and read-only.
Task	Task to be executed: E-execute a statement; S-Scroll one line forward; R-read one attribute; CN-write an attribute during an insert protocol without committing; UN-write an attribute during an update protocol without committing; CC-commit an insert protocol; UC-commit an update protocol; D-delete a row.
JDBC	Time to execute the standard JDBC code
T	Overhead ids
ΔT	Overhead in ns
%	Induced overhead in %, (ΔT/JDBC). δ means that the %<10^{-2}.
OM	ORCA methods involved in the task execution

Table IV. Algorithms for the S_s scenario.

Execute	Scroll	Insert commit	Update commit
1. new table: 1 row	1. new table: 1 row	1. new table: 0 row	1. new table: 1 row
2. start timer	2. select all rows	2. select 0 rows	2. select row
3. create statement	3. start timer	3. start timer	3. start timer
4.exec. statement	4. step one row	4. insert attribute	4. update attribute
5. Stop timer	5. stop timer	5. commit	5. commit
		6. stop timer	6. stop timer
Read	**Delete**	**Insert no commit**	**Update no commit**
1. new table: 1 row	1. new table: 1 row	1. new table: 1 row	1. new table: 1 row
2. select row	2. select all rows	2. select row	2. select row
3. start timer	3. start timer	3. start timer	3. start timer
4. read attribute	4. delete row	4. update attribute	4. update attribute
4. …	5. stop timer	…	…
5. stop timer		5. stop timer	5. stop timer

Table V. Collected measurements for scenarios S_i, S_u and S_d.

Statement	T (ns)	ORCA Overhead		%
		TR	ΔT(ns)	
Insert	419,475	TR_1	3	δ
Update	4,449,553	TR_1	3	δ
Delete	4,399,443	TR_1	3	δ

Table VI. Algorithms for scenarios S_i, S_u and S_d.

Insert	Update	Delete
1. create table:0 row 2. start timer 3. insert one row 4. stop timer	1. create table:1 row 2. start timer 3. update row 4. stop timer	1. create table:1 row 2. start timer 3. delete row 4. stop timer

$V=TR_1+TR_9+TR_4=3+6+6=15$ns. The induced overheads are about 0.004% and 0.4%, respectively. If we consider, which is not true, that execution time of task E does not increase with the number of returned rows, to get an overhead of 1% to read all returned rows, an FR LMS should contain 438 rows for a reused Business Worker and 440 rows for a new Business Worker. New measurements for task E (LMS with 440 rows) were collected to analyze the impact. Task E took 711,481ns. The new computed overhead is now 0.6% for a re-used and for a new Business Worker, which is lower than 1% initially foreseen. Task E for SR, FU and SU LMS is much slower. The induced overhead is about 10times lower. Regarding the update, insert and delete protocols, they use much more CPU time than the read protocol and, therefore, the induced % overhead by JORCA is much lower than the ones just presented. Remember that these protocols require an updatable LMS and to update and delete an attribute it is necessary to execute a Select statement in advance.

Table V presents the collected measurements for scenarios S_i, S_u and S_d. Table VI describes the algorithm used in each task. Once again we are before results that prove that the ORCA induced overheads are very low for the three scenarios. They range between $7x10^{-4}$% and $7x10^{-5}$%. Measurements were also collected for pre-compiled SQL statements. They have not been presented because, for the tasks under evaluation, their values were so close to compiled-on-the-fly SQL statements that they would not bring any novelty.

V. Related Work

Beyond CLI (ODBC, JDBC, ADO.NET[12]), several solutions have been devised to improve the development process of business tiers. From them Object-to-Relational mapping (O/RM) tools [13, 14] (LINQ [15], Hibernate [16], Java Persistent API (JPA) [17]) have had a significant acceptance in the academic and commercial forums. Other solutions, such as embedded SQL [18] (SQLJ [19]), have achieved some acceptance in the past. Others were proposed but without any general known acceptance: Safe Query Objects [20] and SQL DOM [21]. O/RM tools are geared to create, in the object-oriented paradigm, static representation models of relational database schemas. The static model is built in a first stage, eventually by a database administrator, and then programmers start the development process. The basic artifacts of the static representation models are classes (entities) each one representing a database table. Through these entities programmers may read data from tables, update data, insert new data and, finally, delete existing data. To support explicit SQL statements, O/RM tools provide proprietary SQL languages. Despite these advantages, O/RM present some drawbacks, such as: 1) they induce an additional overhead when compared to CLI; 2) they were not devised to support complex SQL statements and, finally, 3) they rely on static models preventing this way an easy process for a dynamic adaptation. Moreover, O/RM do not promote a clear separation of roles. Programmers may use embedded language extensions and other embedded functionalities to extend pre-built static models.

Pereira et al. [22] presented a work in progress in its initial phase with the aim of creating business tier components driven by a single static interface (sort of Business Interface responsible for managing all SQL statements) which is built during the development phase of components. Any modification in this interface entails a new manual development process for the component under maintenance. The approach is aimed at providing a proposal for reusable components statically built to address a specific business area, such as accountability or sales. In [22], no evidence of its feasibility and applicability were presented. The work presented in [23] addresses a similar research question as the one presented in [22]. The main difference is that instead of one single static interface, it supports several static (pre-defined) interfaces. The paper explains the architecture but does not provide any evidence of its feasibility or applicability. It even does not provide any comparative study with other equivalent implementations. Therefore, both works, [22] and [23], do not address the key issue of the work described in this paper, which is related to the proposal of components with the capacity of being dynamically adapted at runtime to address any evolving process of business tiers.

Aspect-oriented programming [24] community considers persistence as a crosscutting concern [25]. Several works have been presented but none addresses the point here under consideration. The following works are emphasized: [26] is focused on separating scattered and tangled code in advanced transaction management; [25] addresses persistence relying on AspectJ; [27] presents AO4Sql as an aspect-oriented extension for SQL aimed at addressing logging, profiling and runtime schema evolution. It would be interesting to see an aspect-oriented approach for the points herein under discussion.

VI. Conclusion

Despite their key advantages, such as SQL expressiveness and SQL performance, a key drawback of CLI has been identified: lack of support to develop business tier components driven by dynamic adaptation. To address this CLI drawback and simultaneously keep their advantages, this paper presented an architecture,

herein known as ORCA. Adaptability of ORCA is based on three pillars: 1) Business Entities are automatically built at runtime from Business Schemas (only IExecutePC, IRead and ISet need to be customized) to address users' needs; 2) each Business Entity is able to manage the execution of any SQL statement whose schema is in accordance with its Business Interface; 3) SQL statements are deployed at runtime to address users' needs. A proof of concept based on Java and JDBC has also been presented (it is also downloaded to be tested and used). To prove that CLI advantages were assured, a performance assessment has been carried out. It showed that for an efficient context, ORCA induced overhead is under 0.004% (reused Business Worker) or 0.4% (new Business Worker, when considering the creation phase) for one row and under 0.6% for 440 rows. The overhead induced in other contexts are at least 10 times lower. Moreover, in real situations, the overhead induced by SQL Server will be higher and, therefore, the percentage overhead induced by ORCA will be even lower. SQL expressiveness was also assured by encoding SQL statements inside strings just as CLI do.

In future work, we plan to address a complementary issue: data security, particularly access control [28]. Client-side access control mechanisms may be implemented by enforcing a deployment policy of SQL statements based on access control policies. Each user is authenticated and SQL statements are deployed in accordance to their profiles. Then, any tentative to execute any unauthorized action is locally prevented. Only authorized actions will be permitted and, therefore, sent to the host database. Beyond access control mechanisms, an access control policy model will also be proposed.

It is expected that the outcome of this research may contribute to open new perspectives on the development process of adaptable components to build business tiers.

REFERENCES

[1] W. Cook, and A. Ibrahim, "Integrating programming languages and databases: what is the problem?," 2011 May: ODBMS.ORG, Expert Article, 2005.

[2] G. T. Heineman, and W. T. Councill, Component-Based Software Engineering: Putting the Pieces Together, 1st ed., p.^pp. 880: Addison-Wesley, 2001.

[3] C. Szyperky, D. Gruntz, and S. Murer, Component Software - Beyond Object-Oriented Programming: Addison-Wesley/ACM Press, 2002.

[4] L. Kung-Kiu, and W. Zheng, "Software Component Models," IEEE Trans. on Soft. Eng., vol. 33, no. 10, pp. 709-724, 2007.

[5] A. Bracciali, A. Brogi, and C. Canal, "A formal approach to component adaptation," Journal of Systems and Software, vol. 74, no. 1, pp. 45-54, 2005.

[6] M. Parsian, JDBC Recipes: A Problem-Solution Approach, NY, USA: Apress, 2005.

[7] Microsoft. "Microsoft Open Database Connectivity," Oct, 2011; http://msdn.microsoft.com/en-us/library/ms710252(VS.85).aspx.

[8] Oracle. "Interface ResultSet," May, 2011; http://download.oracle.com/javase/6/docs/api/java/sql/ResultSet.html.

[9] Microsoft. "RecordSet (ODBC)," 2011 Jun; http://msdn.microsoft.com/en-us/library/5sbfs6f1.aspx.

[10] P. V. Elizondo, and K.-K. Lau, "A Catalogue of Component Connectors to Support Development with Reuse," Journal of Systems and Software, vol. 83, no. 7, pp. 1165-1178, 2010.

[11] Orcale. "Map Interface," 2012 Jan; http://docs.oracle.com/javase/6/docs/api/java/util/Map.html.

[12] G. Mead, and A. Boehm, ADO.NET 4 Database Programming with C# 2010, USA: Mike Murach & Associates, Inc., 2011.

[13] W. Keller, "Mapping Objects to Tables - A Pattern Language," in European Conference on Pattern Languages of Programming Conference (EuroPLoP), Irsse, Germany, 1997.

[14] R. Lammel, and E. Meijer, "Mappings Make data Processing Go 'Round: An Inter-paradigmatic Mapping Tutorial," in Generative and Transformation Techniques in Soft. Eng., Braga, Portugal, 2006.

[15] M. Erik, B. Brian, and B. Gavin, "LINQ: Reconciling Object, Relations and XML in the .NET framework," in ACM SIGMOD International Conference on Management of Data, Chicago,IL,USA, 2006, pp. 706-706.

[16] B. Christian, and K. Gavin, Hibernate in Action: Manning Publications Co., 2004.

[17] D. Yang, Java Persistence with JPA, p.^pp. 390: Outskirts Press, 2010.

[18] J. W. Moore, "The ANSI binding of SQL to ADA," Ada Letters, vol. XI, no. 5, pp. 47-61, 1991.

[19] Part 1: SQL Routines using the Java (TM) Programming Language, 1999.

[20] R. C. William, and R. Siddhartha, "Safe query objects: statically typed objects as remotely executable queries," in 27th Int. Conf. on Software Engineering, St. Louis, MO, USA, 2005, pp. 97-106.

[21] A. M. Russell, and H. K. Ingolf, "SQL DOM: compile time checking of dynamic SQL statements," in 27th Int. Conf. on Software Engineering, St. Louis, MO, USA, 2005, pp. 88-96.

[22] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "A Reusable Business Tier Component with a Single Wide Range Static Interface," in ECSA: 5th European Conference on Software Architecture, Essen, Germany, 2011, pp. 216-219.

[23] O. M. Pereira, R. L. Aguiar, and M. Y. Santos, "An Adaptable Business Component Based on Pre-defined Business Interfaces," in 6th ENASE: Evaluation of Novel Approaches to Software Engineering, Beijing, China, 2011, pp. 92-103.

[24] J. L. Gregor Kiczales, Anurag Mendhekar, Chris Maeda, Cristina Lopes Videira, Jean-Marc Loingtier, Joh Irwin, "Aspect-Oriented Programming," in ECOOP, Jyvaskyla,Finland, 1997, pp. 220-242.

[25] R. Laddad, AspectJ in Action: Practical Aspect-Oriented Programming, Greenwich,CT,USA: Manning Publications, ,2003.

[26] J. Fabry, and T. D'Hondt, "KALA: Kernel Aspect Language for Advanced Transactions," in Proceedings of the 2006 ACM Symposium on Applied Computing, Dijon, France, ,2006, pp. 1615-1620.

[27] T. Dinkelaker, "AO4SQL: Towards an Aspect-Oriented Extension for SQL," in Proceedings of the 8th Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE'11), Zurich, Switzerland, 2011.

[28] P. Samarati, and S. D. C. d. Vimercati, "Access Control: Policies, Models, and Mechanisms," Foundations of Security Analysis and Design (LNCS), vol. 2171, pp. 137-196, 2001.