AQL: A Declarative Artifact Query Language

Maroun Abi Assaf[1(✉)], Youakim Badr[1], Kablan Barbar[2], and Youssef Amghar[1]

[1] University of Lyon, CNRS, INSA-Lyon, LIRIS, UMR5205, 69621 Lyon, France
{maroun.abi-assaf, youakim.badr, youssef.amghar}@insa-lyon.fr
[2] Faculty of Sciences, Lebanese University, Fanar Campus, Jdeidet, Lebanon
kbarbar@ul.edu.lb

Abstract. Business Artifacts have recently emerged as a compelling paradigm to develop data-centric processes, supporting flexible and knowledge intensive business processes. Artifact-centric process models, as an alternative to predefined activity-centric process models, are easy to be understood and managed by non-IT specialists. Artifacts are also complex entities, which include information models, states, services and transition rules. They interact with each other, updating their information models and evolve following their lifecycles. Despite the increasing glamour that was raised on artifacts from research and business communities, the lack of expressive languages to manipulate and interrogate them, limits their widespread usage. In this paper, we define a declarative *Artifact Query Language* (*AQL*) that relies on a relational schema to define, manipulate, and query artifact types. The *AQL* takes full-advantage of the well-established SQL to manipulate the relational schema and relieves casual users from the need to directly deal with SQL's statements and the underlying relational model (i.e., relations, keys constraints, and constructing complex queries).

Keywords: Artifact types · Domain specific languages · Query languages · Compilers · SQL abstraction layer

1 Introduction

Traditionally, business processes have been modeled as workflows of activities. The primary disadvantage of such approach is the separation between data models and process aspects of businesses [5]. An alternative and more recent approach is the artifact-centric process modeling approach [14], which combines both data and their manipulation into cohesive and modular units known as business artifacts or artifact types in a broad sense. The artifact-centric approach demonstrates many advantages and benefits including; enabling a natural modularity and componentization of business processes, facilitating business transformations and organizational changes and providing a framework of varying levels of abstraction to develop business processes to name a few [5, 8]. On the other hand, being complex entities, artifacts require suitable

J. Pokorný et al. (Eds.): ADBIS 2016, LNCS 9809, pp. 119–133, 2016.
DOI: 10.1007/978-3-319-44039-2_9

methods and technologies in order to be implemented and treated efficiently. Nonetheless, artifacts have attracted much attention from the research communities. Few initiatives attempt to manage them recently as graphical-based models (i.e. Artiflow) or as data objects using relational databases (SQL) or data-centric dynamic system (DCDS) [7, 16]. Since artifacts are complex models, including attribute-value pairs information model, state-based lifecycles, and transitions that invoke services to move artifacts from a current state to a new state of their lifecycle. These initiatives show their limits and do not allow end-users to benefit from the full potential and flexibility that artifacts can provide. In fact, graphical-based models often focus on defining and running artifact processes. They are thus not convenient for querying artifacts. However, using relational databases to manage artifact structures require nested and tedious queries taking into account table relationships, constraints, dependencies and their keys. As a result, a declarative and expressive artifact language becomes essential to efficiently manage artifact types. Such language opens an era for using artifacts beyond business processes and builds new class of applications in various domains. For example, artifact types can represent connected devices or urban entities in the context of smart cities.

An artifact specific language should be compatible with the artifact model. Firstly, it should consider that an artifact, as a cohesive entity, could be created, updated or dropped as the need arises. Moreover, artifacts have to interact with each other through events in order to exchange necessary information and update their lifecycles. Secondly, artifacts must evolve in a state-based lifecycle starting at an initial state, passing in intermediate states, and ending in one of their final states. As a result, an artifact specific language should not only meet all these requirements and challenges but it should also be simple enough in order to be used by non-IT specialists within and beyond business processes.

In this paper, we propose the *Artifact Query Language* (*AQL*) that is specifically designed to take full advantage of the artifact model. The *AQL* is a high-level declarative language that deals with defining and manipulating artifacts at the business logic level. It is based-on the SQL and extends it with artifact domain specific statements. The *AQL* relieve users from dealing with multiple tables, primary and foreign keys constraints, and constructing complex SQL queries that include joins and nested sub-queries. As a result, The *AQL* is intended to be used by non-IT specialists and enables them to write queries that focus on the artifact logic instead of dealing with technical details related to SQL and artifact complex structure management. Moreover, the *AQL* can co-exist with graphical based artifact systems such as *Artiflow* [16]. The proposed *AQL* is an abstraction layer over SQL and translates all its queries into underlying SQL queries. The semantics of the AQL is thus expressed in terms of the relational model.

The remaining of the paper is organized as follows. Section 2 describes the syntax of *AQL* and provides query examples. Section 3 presents the semantics of *AQL* expressed in terms of the relational model whereas Sect. 4 illustrates the prototype implementation. Related works and similar initiatives are discussed in Sect. 5. Finally, Sect. 6 concludes the work and provides future perspectives.

2 Syntax

The *Artifact Query Language* (*AQL*) is a high-level language that is based-on the relational database SQL. *S*ince it is an abstraction layer over SQL, it follows the syntax of SQL statements, with some variations, but provides a simplified syntax that is translated into SQL queries. The *AQL* consists of the *Artifact Definition Language* (*ADL*) to define artifact classes, and the *Artifact Manipulation Language* (*AML*) to manage artifact instances.

Thus, *ADL* includes a statement to define artifact classes. For example, the *Create Artifact* statement allows the definition of a list of simple and complex data attributes, references to child artifact classes, and a list of states, representing stages of artifact lifecycles [4]. As for the *AML,* it includes statements to instantiate, manipulate and interrogate artifact instances. For example, the *New* statement instantiates new artifact instances; the *Update* statement updates simple attribute types and states; the *Insert Into* and *Remove From* statements are used to insert and remove (business) *objects* (complex attributes values) and child artifacts (reference attributes values) respectively into and from artifacts; the *Delete* statement deletes artifact instances altogether from the database; the *Retrieve* statement retrieves artifact instances that meet conditions.

In the following sections, we first describe a scenario to illustrate the *AQL* with query examples for each of its statements. We secondly introduce in details the syntax of *ADL* and *AML* statements.

2.1 Example Scenario

In order to illustrate the AQL queries through a scenario, we define business processes related to the candidate admission application in an academic program. In this scenario, the business process in a university begins with the candidate submitting his application to the secretary of the Master program. The secretary creates a new application file to process the candidature and records personnel information such as; first name, last name and age. The secretary then collects and scans required documents including a CV, diplomas, and motivation letters. If all required documents are presented, the secretary marks the application as complete, otherwise the application is marked as incomplete and is rejected. After that, the master program chair inspects all complete applications and checks if they are eligible. If an application is not eligible, the candidature is rejected; otherwise the candidate is selected to be interviewed by academic committee members on a specified date and location. During the interview, notes and decisions about the candidate are taken by the committee members. If needed, additional interviews can also be scheduled for the same candidate. Finally, interviews are evaluated and decisions are made about whether candidates are accepted or rejected.

We identify two artifacts in the candidate admission process; (1) The *Candidate Application Artifact (CAA)*, which deals with processing candidate applications and tracks various decisions made about them, and (2) The *Candidate Interview Artifact (CIA)*, which deals with interviewing candidates, collecting and evaluating interviews' information. In the following sections, we rely on these artifacts to formulate query examples.

2.2 Artifact Definition Language

The *Artifact Definition Language (ADL)* is used to define an artifact class or artifact type with respect to the artifact model. It consists of a list of data attributes and a list of states. Data attributes can be of three types: *simple type*, *complex type*, and *reference type*.

1. *The simple attribute types* represent simple types such as *Boolean*, *Integer*, *Real* or *String*. Simple attribute can only store one value at a time. For example the *FirstName* attribute type in the *Candidate Application Artifact* may have the string value *"John."*
2. *The complex attribute types* represent complex structures that are made up of one or more simple attribute types. These complex structures describe the (business) *objects* that can be inserted and/or removed from artifacts. For example, the *Documents* complex attribute type in the *Candidate Application Artifact* is formed from a tuple of three simple attribute types: *Type, Title,* and *URL.* Complex attribute types have a cardinality of one or many. For example, several *Documents* can be inserted into the *Candidate Application Artifact.*
3. *The reference attribute types* in a master artifact represent references to child artifacts related to the master artifact. Reference attribute types have a cardinality of one or many. In other words, a reference type attribute can store a list of references to several artifact instances. For example, an *Interviews* reference attribute type in the *Candidate Application Artifact* refers to the *Candidate Interview Artifact* and thus, may have a list of one or more references to *Candidate Interview Artifact* instances.
4. In addition, the list of states in the artifact class describes possible stages of the artifact's lifecycle. These states include *initial*, *final*, or *intermediate* states. An artifact instance can only be in one state of its lifecycle at a time. For example, the *Candidate Interview Artifact* instance may have the *accepted* state during its processing.

The *Create Artifact* statement is illustrated in Fig. 1(a) and shows the example of defining the *Candidate Application Artifact* (*CAA*). *ApplicationArtifactId, FirstName, LastName* and *Age* are simple attribute types. *Documents* is a complex attribute type. Whereas *Interviews* is a reference attribute type pointing to the *Candidate Interview Artifact (CIA). Initialized*, *Created*, *Rejected*, *Complete*, *Interviewed*, and *Accepted* denote states of its artifact lifecycle in which *Initialized* is the initial state, *Rejected* and *Accepted* are two final states, and remaining states are intermediate states. Figure 1(b) illustrates the grammar of the *Create Artifact Statement*.

2.3 Artifact Manipulation Language

The *Artifact Manipulation Language* (*AML*) consists of six statements to instantiate, modify and retrieve artifact instances.

```
Create Artifact CAA With
Attributes (
    ApplicationArtifactId : Integer,
    FirstName : String,
    LastName : String,
    Age : Integer,
    Documents : { Type : String,
                  Title : String,
                  URL : String     },
    Interviews : CIA
)
States (
           Initialized as initial state,
           Created,
           Rejected as final state,
           Complete,
           AwaitingInterview,
           Interviewed,
           Accepted, as final state   )
```

a) Create artifact query example

```
CREATEARTIFACT: "Create Artifact " BANAME " With "
          ATTRIBUTECLAUSE STATECLAUSE;
ATTRIBUTECLAUSE: "Attributes (" ATTRIBUTELIST ")";
ATTRIBUTELIST: ATTRIBUTE | ATTRIBUTE "," ATTRIBUTELIST;
ATTRIBUTE: ATTRIBUTENAME ":" ATTRIBUTETYPE;
ATTRIBUTETYPE: SIMPLETYPE | COMPLEXTYPE | REFERENCETYPE;
SIMPLETYPE: "Boolean" | "Integer" | "Real" | "String";
COMPLEXTYPE: "{" ATTRIBUTELIST "}";
REFERENCETYPE: BANAME;
STATESCLAUSE: "States (" STATELIST ")";
STATELIST: STATE | STATE "," STATELIST;
STATE: STATENAME | STATENAME "As Initial State" |
          STATENAME "As Final State";
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
STATENAME: IDENTIFIER;
IDENTIFIER: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

b) Create artifact statement grammar

Fig. 1. Create artifact statement

2.3.1 Instantiate Statement

Since artifacts denote complex data structures that are composed of simple, complex and reference attribute types and a list of states, several tuples must be inserted into two or more tables in the underlying relational database when creating new artifact instances. The traditional SQL's INSERT statement is thus not sufficient to create several tuples. Hence, the *New* statement instantiate a new artifact instance and initializes its attributes values and state.

The *New* statement exhibits several modes of uses. The first mode creates a new artifact instance and initializes some of its simple attributes as illustrated in Fig. 2(1) where a *Candidate Application Artifact* instance is created with *100543* as the value of its *ApplicationArtifactId* attribute. Additionally, its state is automatically initialized to its initial state *"initialized"* as defined in the *Create Artifact* query in Fig. 1.

```
1)  New CAA With
    Values(100543)
2)  New CAA With
    Values(100543, "John", "Smith", 23)
    Set State To Created
3)  New CAA With
    Values(100543, "John", "Smith", 23)
    Documents {
      ("CV", "Curriculum Vitae", "http://..."),
      ("Diploma", "Bachelor in CS", "http://..."),
      ("Letter", "Recom. Letter", "http://...")
    }
    Set State To Submitted
4)  New CAA With
    Values(100543, "John", "Smith", 23)
    Documents {
      ("CV", "Curriculum Vitae", "http://..."),
      ("Diploma", "Bachelor in CS", "http://..."),
       ("Letter", "Recom. Letter", "http://...")
    }
    Interviews having ( InterviewArtifactId = 205465 )
    Interviews having ( InterviewArtifactId = 206721 )
    Set State To AwaitingInterview
```

Fig. 2. New query examples

In order to initialize the artifact to a particular state, the "***Set State To*** *StateName"* clause must be used as illustrated in Fig. 2(2) where in addition to initializing the *ApplicationArtifactId*, *FirstName*, *LastName* and *Age*, the state is initialized to *"Created"*. The *New* statement can also be used to initialize complex attributes as illustrated in in Fig. 2(3) where three documents including a CV, a diploma, and a recommendation letter are inserted into the new *Candidate Application Artifact* instance. Finally, the *New* statement can be used to initialize reference attributes as illustrated in Fig. 2(4)

```
NEW: "New" BANAME "With"
     SIMPLEATTCLAUSE COMPLEXATTCLAUSE? REFERENCEATTCLAUSE? STATECLAUSE?;
SIMPLEATTCLAUSE: "Values (" CONSTANTVALUELIST ")";
CONSTANTVALUELIST: CONSTANTVALUE | CONSTANTVALUE "," CONSTANTVALUELIST;
COMPLEXATTCLAUSE: COMPLEXATTRIBUTELIST
COMPLEXATTRIBUTELIST: COMPLEXATTRIBUTE | COMPLEXATTRIBUTE " " COMPLEXATTRIBUTELIST;
COMPLEXATTRIBUTE: ATTRIBUTENAME "(" TUPLELIST ")";
TUPLELIST: TUPLE | TUPLE "," TUPLELIST;
TUPLE: "(" CONSTANTVALUELIST ")";
REFERENCEATTCLAUSE: REFERENCEATTRIBUTELIST;
REFERENCEATTRIBUTELIST: REFERENCEATTRIBUTE |
     REFERENCEATTRIBUTE  " " REFERENCEATTRIBUTELIST;
REFERENCEATTRIBUTE: ATTRIBUTENAME "having (" CONDITION ")";
CONDITION: CONDITIONPREDICATELIST;
CONDITIONPREDICATELIST: CONDITIONPREDICATE |
           CONDITIONPREDICATE "And" CONDITIONPREDICATELIST;
CONDITIONPREDICATE: ATTRIBUTENAME PREDICATEOP CONSTANTVALUE;
PREDICATEOP: "=" | "<" | ">" | "<=" | ">=";
STATECLAUSE: "Set State To" STATENAME;
BANAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER | CONSTANTVALUE DIGIT;
IDENTIFIER: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Fig. 3. New statement grammar

where two references to *Candidate Interview Artifact* instances with *InterviewArtifactId* respectively equal to *205465* and *206721* are inserted into the new *CandidateApplicationArtifact* instance. Figure 3 illustrates the grammar of the *New* statement.

2.3.2 Modification Statements

Modification of artifact instances can be performed at several levels: (1) *update* simple attribute values, (2) *update* states, (3) *update* tuples of complex attributes, (4) *insert* or *remove* tuples of complex attributes, (5) *insert* or *remove* references to child artifacts, and finally (6) *delete* artifact instances.

First, simple attribute values of artifact instances can be updated as in SQL using the *Update* statement as illustrated in Fig. 4(1). Similarly, the states of artifact instances can be updated using the *Update* statement as illustrated in Fig. 4(2).

```
1) Update CAA
   Set FirstName = "Johny", Age = 24
   Where ApplicationArtifactId = 100543
2) Update CAA
   Set State to Rejected
   Where ApplicationArtifactId = 100544
3) Update Documents In CAA
   Set Documents.Type = "Certificate"
   Where CAA.ApplicationArtifactId=100543
   And Documents.Title = "Bachelor in CS"
4) Insert Documents  Into CAA
   {
    ("Diploma", "Bachelor in CC", "http://..."),
    ("Letter", "Motivation Letter", "http://...")
   }
   Where ApplicationArtifactId = 100543
5) Insert Interviews Into CAA
   Where CAA.ApplicationArtifactId = 100543
   And Interviews.InterviewArtifactId = 654321
6) Remove Documents From CAA
   Where CAA.ApplicationArtifactId=100543
   And Documents.title = "Bachelor in Computer Science"
7) Remove Interviews From CAA
   Where CAA.ApplicationArtifactId=100543
   And Interviews.InterviewArtifactId = 206721
8) Delete CAA
   where ApplicationArtifactId = 100543
```

Fig. 4. Modification query examples

In this case, the "***Set State To*** *StateName"* clause is used to specify the new state. Finally, modifications of tuples of complex attributes are also performed using the *Update* statement expressed with the *"**Update** AttributeName **In** ArtifactName"* clause

to indicate in which artifact the complex attribute is located. Figure 4(3) illustrates an example where the *Type* attribute of the document with the title *"Bachelor in CS"* in the *Candidate Application Artifact* instance (id *100543*) is updated with the value *"Certificate"*.

Inserting tuples of complex attributes into artifact instances can be performed using the *"**Insert** AttributeName **Into** ArtifactName"* clause to indicate in which artifact the complex attribute is located and specifying a list of tuples to be inserted (see Fig. 4(4)). Similarly, inserting a reference into a child artifact in a given artifact can be performed using the *Insert Into* statement (Fig. 4(5)). In this case the child artifact instance is selected using the condition specified in the *"**Where** Condition"* clause. Removing complex attribute tuples and child artifact references from artifact instances can be performed using the *Remove From* statement as illustrated in Fig. 4(6) and 4(7). The *Remove From* statement functions in the same way as the *Insert Into* statement. Finally, deletion of artifact instances can be performed using the *Delete* statement as illustrated in Fig. 4(8). In this case, the artifact instance including its complex attributes tuples and child artifact references are deleted. Figure 5 illustrates the grammar of modification statements where the production rules for the ***WHERECLAUSE*** are omitted and listed instead in Fig. 7 for readability concerns.

```
UPDATE: "Update" ( BANAME | ATTRIBUTENAME "in" BANAME ) SETCLAUSE WHERECLAUSE;
SETCLAUSE: SETSTATE | SETATTRIBUTES;
SETSTATE: "Set State To" STATENAME;
SETATTRIBUTES: "Set" ATTRIBUTEASSIGNMENTLIST;
ATTRIBUTEASSIGNMENTLIST: ATTRIBUTEASSIGNMENT | ATTRIBUTEASSIGNMENT "," ATTRIBUTEASSIGNMENTLIST;
ATTRIBUTEASSIGNMENT: ATTRIBUTENAME "="  CONSTANTVALUE;
INSERT: "Insert" ATTRIBUTENAME "Into" BANAME COMPLEXATTCLAUSE? WHERECLAUSE;
COMPLEXATTCLAUSE: "{" TUPLELIST "}";
REMOVE: "Remove" ATTRIBUTENAME "From" BANAME WHERECLAUSE;
DELETE: "Delete" BANAME WHERECLAUSE;
BANAME: IDENTIFIER;
STATENAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER | CONSTANTVALUE DIGIT;
IDENTIFIER: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Fig. 5. Modification statements grammar

2.3.3 Retrieve Statement

Artifact instances and their content can be retrieved using the *Retrieve* statement, which is an abstraction statement over SQL's SELECT statement. Retrieving artifact instances according to the values of their simple attributes and state is performed as illustrated in Fig. 6(1). All information related to the artifact instance including the values of its simple attributes, state, tuples of its complex attributes, and artifact instances of its reference attributes are retrieved by default. The *"**Only**"* keyword restricts the retrieval of values to simple attributes and states of the master artifact (see Fig. 6(2)). Retrieving artifact instances according to the values of their complex attributes is performed using the *"**Include**"* operator as illustrated in Fig. 6(3). The asterisk symbol (*) is used to match any string of characters. In this case, the retrieved artifact instances should have two documents with the *Title* respectively equal to *"Bachelor in Computer Science"* and *"Recommendation Letter from Professor"*. Retrieving artifact instances according

```
1) Retrieve CAA
   Where State is Accepted
   And Age = 23
2) Retrieve Only CAA
   Where State is Accepted
   And Age = 23
3) Retrieve CAA
   Where State Is Submitted
   And age = 23
   And Documents Include {
    ( * , "Bachelor in Computer Science", * ),
    ( * , "Recommendation Letter from Professor", *)
   }
4) Retrieve CAA
   Where State Is AwaitingInterview
   And age = 23
   And Interviews Having ( Date = CURRENTDATE
                           And State Is Ready )
5) Retrieve Documents From CAA
   Where CAA.ApplicationArtifactId=100543
6) Retrieve Interviews From CandidateApplicationArtifact
   Where CAA.ApplicationArtifactId=100543
```

Fig. 6. Retrieve query examples

to their child artifacts is performed as illustrated in Fig. 6(4). In this case, the *"**Having**"* operator is used to specify the condition that the child artifacts should meet. Finally, retrieving only the values of complex or reference attributes can be achieved using the *"**Retrieve** AttributeName **From** ArtifactName"* clause (see Fig. 6(5) and (6)).

Figure 7 illustrates the grammar of the *Retrieve* statement.

```
RETRIEVE: "Retrieve" ( ("Only")? BANAME | ATTRIBUTENAME "From" BANAME) WHERECLAUSE;
WHERECLAUSE: "Where" WHEREPREDICATELIST;
WHEREPREDICATELIST: WHEREPREDICATE | WHEREPREDICATE "AND" WHEREPREDICATELIST;
WHEREPREDICATE: STATEPREDICATE | NULLPREDICATE | NOTNULLPREDICATE |
          COMPARISIONPREDICATE | INCLUDEPREDICATE | HAVINGPREDICATE;
STATEPREDICATE: "State is" STATENAME;
NULLPREDICATE: ATTRIBUTEIDENTIFICATION "Is Null";
NOTNULLPREDICATE: ATTRIBUTEIDENTIFICATION "Is Not Null";
COMPARISIONPREDICATE: ATTRIBUTEIDENTIFICATION PREDICATEOP CONSTANTVALUE;
PREDICATEOP: "=" | "<" | ">" | "<=" | ">=";
INCLUDEPREDICATE: ATTRIBUTENAME "Include {" TUPLELIST "}";
TUPLELIST: TUPLE | TUPLE "," TUPLELIST;
TUPLE: "(" CONSTANTVALUELIST ")";
CONSTANTVALUELIST: CONSTANTVALUE | CONSTANTVALUE "," CONSTANTVALUELIST;
HAVEPREDICATE: ATTRIBUTENAME "Having (" CONDITION ")";
CONDITION: CONDITIONPREDICATELIST;
CONDITIONPREDICATELIST: CONDITIONPREDICATE |CONDITIONPREDICATE "And" CONDITIONPREDICATE;
CONDITIONPREDICATE: WHEREPREDICATE;
ATTRIBUTEIDENTIFICATION: ((BANAME | ATTRIBUTENAME) ".")? ATTRIBUTENAME;
BANAME: IDENTIFIER;
STATENAME: IDENTIFIER;
ATTRIBUTENAME: IDENTIFIER;
CONSTANTVALUE: LETTER | DIGIT | CONSTANTVALUE LETTER | CONSTANTVALUE DIGIT;
IDENTIFIER: LETTER | IDENTIFIER LETTER | IDENTIFIER DIGIT;
LETTER: "a" ... "z" | "A" ... "Z";
DIGIT: "0" ... "9";
```

Fig. 7. Retrieve statement grammar

3 AQL Semantics

This section defines the semantics of *AQL* in terms of the *Relational Model*. Firstly we formalize the notion of an *artifact class* based on [4] and secondly we describe every *AQL* statement with its operational semantics using relational model concepts as described in [2].

We start by assuming the existence of the following pairwise disjoint countably infinite sets: $\mathcal{D}$ for constants; i.e. data values. $\mathcal{C}$ of artifact names. $\mathcal{A}$ of attribute names. STS of artifact states. $\mathcal{T}_{\text{prim}}$ of primitive types, including *Boolean*, *Integer*, *Real* or *String*. $\mathcal{T}_{\text{com}}$ of complex types, where elements of $\mathcal{T}_{\text{com}}$ are subsets of $\mathcal{A}$, and $\mathcal{T}$ of types, where $\mathcal{T} = \mathcal{T}_{\text{prim}} \cup \mathcal{T}_{\text{com}} \cup \mathcal{C}$.

We also give some simple notations for *relations* and *relation schemas*. For a given relation schema R, we denote by $schema(R) \subseteq \mathcal{A}$ the set of attributes in R. The primary key of R is denoted by $key(R) \subseteq schema(R)$. A tuple t over R is an element of $\mathcal{D}^{|schema(R)|}$, and a relation r over R is a finite set of tuples over R such that $r \subseteq D^{|schema(R)|}$. We also assume the existence of a relation *states* over a relation schema *States* used to store information about states of lifecycles with $schema(States) = \{Artifact, State, Type\}$ and $key(States) = \{Artifact, State\}$.

We also make use of the following *relational algebra* operators; *selection*, *projection*, *cartesian product* and *assignment*. Selection is denoted by $\sigma_c(r)$ where a subset of tuples that meet condition c is selected from the relation r. Projection is denoted by $\pi_{a1,\ldots,an}(r)$ where the result is a relation of n attributes obtained by erasing from the relation r the attributes that are not listed in $a_1,\ldots,a_n$. Cartesian product is denoted by $r_1 \times r_2$ where the result is a relation that combines r_1 and r_2. Relational algebra expressions can be constructed using selection, projection and Cartesian product operators in addition to mathematical union and set difference operators. *Assignment* is denoted by $r \leftarrow E$ where the result of the relational algebra expression E is assigned to the relation r. Using the assignment operator, we can define *insert*, *delete* and *update* operations on relations. Inserting a tuple t into a relation r is defined as $r \leftarrow r \cup t$. Deleting a tuple t from a relation r is defined as $r \leftarrow r - t$. Updating a tuple t in a relation r is defined as $r \leftarrow r - t \cup t'$ where t' is the updated tuple.

3.1 Artifact Definition Language

The *Create Artifact* statement of *ADL* is used to define artifact classes according to the structure defined in Definition 1.

Definition 1 (Artifact Class). An *Artifact Class* C is a tuple (C, A, τ, Q, s, F) where $C \in \mathbf{C}$ is a class name, $A \subseteq \mathcal{A}$ is a finite set of attributes, $\tau: A \rightarrow \mathcal{T}$ is a total mapping, $Q \subseteq$ STS is a finite set of states, and $s \in Q$, $F \subseteq Q$ are respectively initial and final states.

Taking as an example the *Create Candidate Application Artifact* query of Fig. 1, we would have: $C = CAA$, $A = \{ApplicationArtifactId, FirstName, LastName, Age, Documents, Interviews\}$, $\tau(ApplicationArtifactId) = Integer$, $\tau(FirstName) = String$, $\tau(LastName) = String$, $\tau(Age) = Integer$, $\tau(Documents) = \{Type, Title, URL\}$ where $\tau(Type) = String$, $\tau(Title) = String$ and $\tau(URL) = String$, $\tau(Interviews) = CIA$, $Q = \{Initialized, Created, Rejected, Complete, AwaitingInterview, Interviewed, Accepted\}$, $s = Initialized$, and finally, $F = \{Rejected, Accepted\}$.

The defined artifact is implemented in the relational model according to the following semantics:

First, a relation schema C_r that represents the artifact class C is created. C_r will contain the simple attributes of C such that $schema(C_r) = \{a \mid a \in A \text{ and } \tau(a) \in \mathcal{T}_{\text{prim}}\}$. In addition to two more attributes: $a_{pk} = concat(C, \text{"\_PK"})$ is the *primary key* of C_r such that $key(C_r) = a_{pk}$, and $a_{st} = State$ is the current state of the artifact. In our

example, we obtain the relation schema *CAA(CAA_PK, ApplicationArtifactId, FirstName, LastName, Age, State)*.

Second, for every complex attribute a_{com} such that $a_{com} \in A$ and $\tau(a_{com}) \in \mathcal{T}_{\text{com}}$, we create an associated relation schema A_r containing the simple attributes constituting a_{com} such that $schema(A_r) = \{a \mid a \in \tau(a_{com})\ and\ \tau(a) \in \mathcal{T}_{\text{prim}}\}$. Additionally, $schema(A_r)$ will contain a primary key attribute a_{pk} such that $key(A_r) = a_{pk}$ and $a_{pk} = concat(a_{com}, \text{"\_PK"})$. Moreover, $schema(A_r)$ will also contain a reference to the artifact in the form of a foreign key a_{fk} of C_r such that $a_{fk} = concat(C_r, \text{"\_FK"})$. In our example, we obtain the relation schema *Documents(Documents_PK, CAA_FK, Type, Title, URL)*.

Third, for every reference attribute a_{ref} of C such that $a_{ref} \in A$ and $\tau(a_{ref}) \in \text{C}$, we create an associated relation schema A_r that contains the foreign keys of the parent and child artifacts such that $schema(A_r) = \{a_{parent}, a_{child} \mid a_{parent} = concat(C, \text{"\_PFK"})\ and\ a_{child} = concat(\tau(a_{ref}), \text{"\_CFK"})\}$. Additionally, both foreign keys will form the primary key of A_r such that $key(A_r) = \{a_{parent}, a_{child}\}$. In our example, we obtain the relation schema *Interviews(CAA_PFK, CIA_CFK)* which is used to store *many-to-many* references between *Candidate Application Artifacts* and *Candidate Interview Artifacts*.

Finally, for every state q of C, we insert a tuple t into the relation *states* such that; *1)* $states \leftarrow states \cup \{(C, q, \text{"default"})\}$ if $q \in Q\ and\ q \neq s\ and\ q \notin F$. *2)* $states \leftarrow states \cup \{(C, q, \text{"initial"})\}$ if $q \in Q\ and\ q = s$. *3)* $states \leftarrow states \cup \{(C, q, \text{"final"})\}$ if $q \in Q\ and\ q \in F$.

3.2 Artifact Manipulation Language

We now describe the semantics of *AML*.

(1) The *new* statement instantiate artifact instances by inserting necessary tuples into the different relations constituting the artifact. The first insert operation inserts a tuple with values of simple attributes and artifact state into the corresponding artifact relation: $artifact \leftarrow artifact \cup \{(k_{parent}, v_1, \ldots, v_n, state)\}$ where k_{parent} is the primary key of the artifact. If the state is not specified in the query, the initial state of the artifact is retrieved and used from the *states* relation using the expression: $\pi_{\text{State}}(\sigma\text{Artifact}_{=artifactname} \wedge \text{Type}=\text{'initial'}(states))$. Similarly, if the state is specified in the query, it is validated using the expression: $\sigma_{\text{Artifact}=artifactname} \wedge \text{State}=statename(states)$. Then, for every complex attribute tuple, an insert operation is performed on the corresponding complex attribute relation: $att_{complex} \leftarrow att_{complex} \cup \{(k_{att}, k_{parent}, v_1, \ldots, v_n)\}$ where k_{att} is the primary key of the inserted tuple and k_{parent} is the foreign key of the parent artifact. Similarly, for every reference attribute value, an insert operation is performed on the corresponding reference attribute relation: $att_{reference} \leftarrow att_{reference} \cup \{(k_{parent}, k_{child})\}$. In this case, k_{parent} is the foreign key of the parent artifact and k_{child} is the foreign key of the child artifact. k_{child} is retrieved according to the specified condition using the expression: $\pi_{\text{Artifact\_PK}}(\sigma_{\text{condition}}(artifact))$.

(2) The *update* statement updates simple attributes of artifacts and complex attributes, in addition to the states of artifacts. First, updating simple attributes and states of artifacts is performed by retrieving the required tuple from the artifact relation using a selection operation: $t \leftarrow \sigma_{condition}(artifact)$ where *condition* is the condition specified in the query. Then, an update operation is performed on the artifact relation: $artifact \leftarrow artifact - t \cup t'$ where t' is the updated tuple. On the other hand, updating complex attributes requires a Cartesian product operation in order to retrieve the correct tuple from the complex attribute relation: $t \leftarrow \pi_{\text{schema}(\mathbf{att}\text{complex})}(\sigma_{\text{condition}\wedge\text{Artifact\_PK=Artifact\_FK}}(artifact \times att_{complex})$. Then, an update operation can be performed on the complex attribute relation: $att_{complex} \leftarrow att_{complex} - t \cup t'$ where t' is the updated tuple.

(3) The *insert* statement inserts tuples into complex or reference attributes relations. First, inserting a tuple $(v_1,...,v_n)$ into a complex attribute is performed by retrieving the primary key of the correct artifact using a projection and selection operations: $k_{parent} \leftarrow \pi_{\text{Artifact\_PK}}(\sigma_{\text{condition}}(artifact))$. Then, an insert operation is performed on the complex attribute relation as follow: $att_{complex} \leftarrow att_{complex} \cup \{(k_{att}, k_{parent}, v_1,...,v_n)\}$. Similarly, inserting a tuple into a reference attribute is performed by retrieving both primary keys of the parent and child artifacts using projection and selection operations: $k_{parent} \leftarrow \pi_{\text{Artifact\_PK}}(\sigma_{\mathbf{c}\text{parent}}(artifact))$ where c_{parent} is the condition related to the parent *artifact.* And $k_{child} \leftarrow \pi_{\text{Artifact\_PK}}(\sigma_{\mathbf{c}\text{child}}(artifact))$ where c_{child} is the condition related to the child *artifact.* Then, an insert operation is performed on the reference attribute relation as follow: $att_{reference} \leftarrow att_{reference} \cup \{(k_{parent}, k_{child})\}$.

(4) The *remove* statement deletes tuples from complex or reference attribute relations. Removing a tuple t from a complex attribute relation is performed similarly to the *update* statement for complex attributes. But, a delete operation is used instead of an update operation: $att_{complex} \leftarrow att_{complex} - t$. On the other hand, removing a tuple from a reference attribute relation is performed similarly to the *insert* statement for reference attributes. But, a delete operation is used instead of an insert operation: $att_{reference} \leftarrow att_{reference} - \{(k_{parent}, k_{child})\}$.

(5) The *delete* statement deletes tuples from artifact relations, in addition to all related tuples from complex and reference attribute relations. First, all tuples from all complex and reference attribute relations are deleted as described in the *remove* statement. Then similarly, the tuple corresponding to the artifact is deleted from the artifact relation.

(6) The *retrieve* statement selects tuples that meet certain conditions from artifact relations, in addition to related tuples from complex and child artifact relations. First, tuples from the artifact relation that meet the condition on simple attributes and state of the artifact are selected using: $r_1 \leftarrow \sigma_{\mathbf{c}\text{parent}}(artifact)$ where c_{parent} is the condition related to the simple attributes and state of the artifact. Second, for conditions on the complex attributes of the artifact, expressed using the *"include"* keyword, further selections are performed on the Cartesian product of r_1 and the related complex attribute relation $att_{Complex}$ such as: $\sigma_{\mathbf{c}\text{complex}\wedge\text{Artifact\_PK=Artifact\_FK}(r1} \times att_{Complex})$ where $att_{Complex}$ is the complex attribute relation, and $\mathbf{c}_{\text{complex}}$ is the condition related to the complex attribute. Similarly, for conditions on the reference attributes of the artifact, expressed using the *"having"* keyword,

a selection is performed on the Cartesian product of r_1, the reference attribute relation $att_{reference}$, and the artifact relation *artifact*: $\sigma_{\text{child} \wedge r1.\text{Artifact\_PK}=\text{Artifact\_PFK} \wedge \text{Artifact\_CFK}=artifact.\text{Artifact\_PK}}(r_1 \times att_{reference} \times artifact)$.

4 Implementation

Using the semantics described in Sect. 3, we have implemented a compiler that translates *AQL* into SQL. The compiler relies on the *AQL* grammar described in Sect. 2 and an extended attribute grammar that uses *synthesized* and *inherited* attributes to generate SQL queries from *AQL* queries. Figure 8 illustrates an example of an *AQL* production rule where *AttName, AttType, AList, RefAtt, MetaType, Sal* and *Sql* are *synthesized* attributes and *ArtName* is an *inherited* attribute. In this production rule several cases exist. (1) If the data attribute has simple type *MetaType (ATTRIBUTETYPE)* ==*"simple"*, then it is appended to a list of simple type data attributes *Sal(ATTRIBUTE)*. (2) If the data attribute has complex type *MetaType (AttributeType)* ==*"complex"*, then its CREATE TABLE SQL query is generated and assigned to *Sql(ATTRIBUTE)*. (3) Similarly, if the data attribute has reference type *MetaType(AttributeType)* ==*"reference"*, then its CREATE TABLE SQL query is generated and assigned to *Sql(ATTRIBUTE)*.

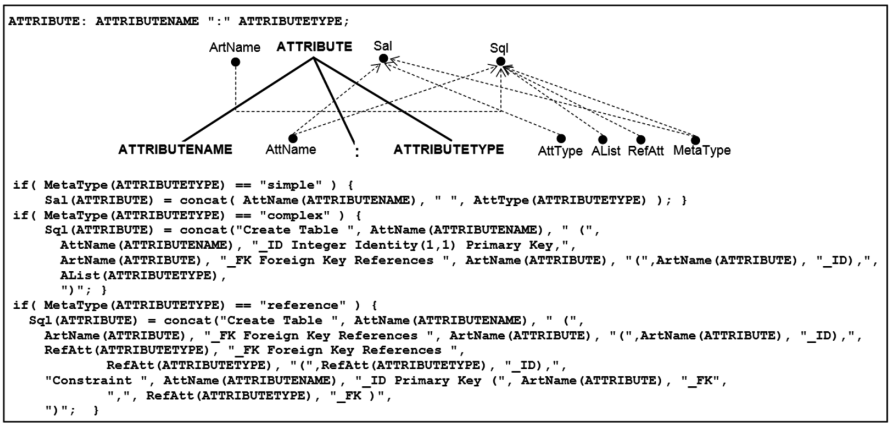

Fig. 8. Attribute grammar example

The compiler relies on the Java Xtext framework to develop our domain-specific language and conduct lexical and syntax analysis and code generation. It connects to a MySQL server as a back-end database. The compiler interface translates queries written in *AQL* into SQL and then executes them.

5 Related Works

Artifacts have gained a lot of attention from a theoretical perspective to formally defining artifacts and studying their properties. Many works have tackled challenges related to lifecycle modeling, conformance, validation, verification, operational semantics and synthesis problems [4, 5, 8]. However, there is still a lot of room for developing artifact-based management systems. The SQL for Business Artifacts (BASQL) introduced in [10] was a first attempt to describe SQL-like statements to define and manipulate artifacts. BASQL still treats business artifacts as traditional relations made of simple type attributes, and as such, instances are manipulated and interrogated using normal SQL statements, operating on relations. On the other hand, many works have focused on defining syntactical and graphical languages to define artifact processes. Works in [13] have introduced the Business Entities and Business Entity Definition Language (BEDL). The BEDL is an XML-based language that specifies business artifact process models, including, Business Entities (or Artifacts), Lifecycles, Access Policies, and Notifications. The BEDL only deals with defining business artifact processes and does not introduce statements to manipulating or interrogating business artifact instances. Business artifact processes are also defined using Active XML (AXML) [1, 3]. A business artifact instance is written as an XML document with embedded function calls. The business artifact process is thus executed by invoking embedded functions and assigning their results to business artifact attributes. The AXML artifact model is concerned with defining and executing the artifact process and does not deal with manipulating and interrogating business artifact instances. Several graphical languages and notations have been developed to define business artifact processes. Authors in [12, 14] introduce a graphical notation to model business artifact lifecycles as finite-state machines. This graphical notation is based on three modeling constructs: Task, Repository, and Flow Connectors. A similar notation is introduced in [11] where the artifact-centric model is called Artifact Conceptual Flow or ArtiFlow (named EZ-Flow in [15]). On the other hand, business artifact lifecycles are declaratively modeled using the Guard-Stage-Milestone (GSM) notations [6, 9]. By using Guards, Stages and Milestones as modeling primitives, the GSM notation allows parallelism and hierarchies in business artifact lifecycles. Roughly speaking, graphical languages and notations focus on defining and executing business artifact processes but they do not include statements to specifically manage business artifact instances. To the best of our knowledge, no work, prior to this work, has focused on defining a declarative language that specifically manipulates and interrogates artifacts with focus on the artifact model regardless its underlying data and structure.

6 Conclusion

Artifacts, as a process modeling approach, advocate the unification of data and processes and offer many advantages to their users. Despite recent advances in the field of artifacts, defining, manipulating and interrogating artifacts are still in their infancy. In this paper, we presented the *Artifact Query Language (AQL)* that seeks to define, manipulate, and interrogate artifacts with declarative SQL-like statements. Future works include the

addition of statements to create business rules and services in *AQL* and the automatic generation of services' method stubs in a procedural programming language. In order to support Artifact streams, we are seeking to extend the AQL with continuous querying capabilities with sliding windows and apply them to high throughput real-time streams in the context of smart cities.

Acknowledgments. This work is generously supported by the 2015 COOPERA funding program of the Rhône-Alpes Region.

References

1. Abiteboul, S., Bourhis, P., Galland, A., Marinoiu, B.: The AXML artifact model. The 16th International Symposium on Temporal Representation and Reasoning, pp. 11–17 (2009)
2. Abiteboul, S., Hull, R., Vianu, V.: Foundations of Databases, vol. 8. Addison-Wesley, Reading (1995)
3. Abiteboul, S., Segoufin, L., Vianu, V.: Modeling and verifying active XML artifacts. IEEE Data Engineering Bulletin **32**(3), 10–15 (2009)
4. Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J.: Towards formal analysis of artifact-centric business process models. In: Alonso, G., Dadam, P., Rosemann, M. (eds.) BPM 2007. LNCS, vol. 4714, pp. 288–304. Springer, Heidelberg (2007)
5. Cohn, D., Hull, R.: Business artifacts: A data-centric approach to modeling business operations and processes. Bulletin IEEE Comput. Soc. Techn. Committee Data Eng. **32**(3), 3–9 (2009)
6. Damaggio, E., Hull, R., Vaculín, R.: On the equivalence of incremental and fixpoint semantics for business artifacts with Guard–Stage–Milestone lifecycles. Inf. Syst.l **38**(4), 561–584 (2013)
7. Heath III, F(., Boaz, D., Gupta, M., Vaculín, R., Sun, Y., Hull, R., Limonad, L.: Barcelona: A design and runtime environment for declarative artifact-centric bpm. In: Basu, S., Pautasso, C., Zhang, L., Fu, X. (eds.) ICSOC 2013. LNCS, vol. 8274, pp. 705–709. Springer, Heidelberg (2013)
8. Hull, R.: Artifact-centric business process models: brief survey of research results and challenges. In: Meersman, R., Tari, Z. (eds.) OTM 2008, Part II. LNCS, vol. 5332, pp. 1152–1163. Springer, Heidelberg (2008)
9. Hull, R., Damaggio, E., De Masellis, R., Fournier, F., Gupta, M., Heath III, F.T., Hobson, S., Linehan, M., Maradugu, S., Nigam, A., Sukaviriya, P.N.: Business artifacts with Guard-Stage-Milestone lifecycles: Managing artifact interactions with conditions and events. In: Proceedings of the 5th ACM International Conference on Distributed Event-based System, pp 51–62 (2011)
10. Joseph, H.R., Badr, Y.: Business artifact modeling: A framework for business artifacts in traditional database systems. In: Enterprise Systems Conference (ES 2014), pp. 13–18 (2014)
11. Liu, G., Liu, X., Qin, H., Su, J., Yan, Z., Zhang, L.: Automated realization of business workflow specification. In: Dan, A., Gittler, F., Toumani, F. (eds.) ICSOC/ServiceWave 2009. LNCS, vol. 6275, pp. 96–108. Springer, Heidelberg (2010)
12. Liu, R., Bhattacharya, K., Wu, F.Y.: Modeling business contexture and behavior using business artifacts. In: Krogstie, J., Opdahl, A.L., Sindre, G. (eds.) CAiSE 2007 and WES 2007. LNCS, vol. 4495, pp. 324–339. Springer, Heidelberg (2007)

13. Nandi, P., Koenig, D., Moser, S., Hull, R., Klicnik, V., Claussen, S., Kloppmann, M., Vergo, J.: Data4BPM, Part 1: Introducing Business Entities and the Business Entity Definition Language (BEDL). IBM Corporation, Riverton (2010)
14. Nigam, A., Caswell, N.S.: Business artifacts: An approach to operational specification. IBM Syst. J. **42**(3), 428–445 (2003)
15. Xu, W., Su, J., Yan, Z., Yang, J., Zhang, L.: An artifact-centric approach to dynamic modification of workflow execution. In: Meersman, R., Dillon, T., Herrero, P., Kumar, A., Reichert, M., Qing, L., Ooi, B.-C., Damiani, E., Schmidt, D.C., White, J., Hauswirth, M., Hitzler, P., Mohania, M. (eds.) OTM 2011, Part I. LNCS, vol. 7044, pp. 256–273. Springer, Heidelberg (2011)
16. Zhao, D., Liu, G., Wang, Y., Gao, F., Li, H., Zhang, D.: A-Stein: A prototype for artifact-centric business process management systems. International Conference on Business Management and Electronic Information **1**, 247–250 (2011)