CREOLE: a Universal Language for Creating, Requesting, Updating and Deleting Resources *

Mayleen Lacouture
Ecole des Mines de Nantes
France
mayleen.lacouture@mines-nantes.fr

Hervé Grall
Ecole des Mines de Nantes
France
herve.grall@mines-nantes.fr

Thomas Ledoux
INRIA Rennes-Bretagne Atlantique
France
thomas.ledoux@inria.fr

In the context of Service-Oriented Computing, applications can be developed following the REST (Representation State Transfer) architectural style. This style corresponds to a resource-oriented model, where resources are manipulated via CRUD (Create, Request, Update, Delete) interfaces. The diversity of CRUD languages due to the absence of a standard leads to composition problems related to adaptation, integration and coordination of services. To overcome these problems, we propose a pivot architecture built around a universal language to manipulate resources, called CREOLE, a CRUD Language for Resource Edition. In this architecture, scripts written in existing CRUD languages, like SQL, are compiled into CREOLE and then executed over different CRUD interfaces. After stating the requirements for a universal language for manipulating resources, we formally describe the language and informally motivate its definition with respect to the requirements. We then concretely show how the architecture solves adaptation, integration and coordination problems in the case of photo management in Flickr and Picasa, two well-known service-oriented applications. Finally, we propose a roadmap for future work.

1 Introduction

The growth of Internet has extended the scope of software applications, leading to Service-Oriented Computing (SOC): it is a new computing paradigm that utilizes services as the basic construct to develop distributed applications, even in heterogeneous environments. To date, there are two popular – and often antagonistic – models for service-oriented computing [22], which we now describe as a process-oriented model and a resource-oriented one.

First, interoperability and integration issues have led to the development of WS-* services technology, mainly based on XML and SOAP. Upon services, which group together operations, processes are defined with orchestration languages, like the Business Process Execution Language

*This work has been partially supported by the CESSA project (http://cessa.gforge.inria.fr/doku.php).

Submitted to:
FOCLASA 2010

for Web Services (BPEL), which is a standard. As processes are central in this model, we say that this model is process-oriented.

More recently, an alternative solution has emerged thanks to its simplicity: RESTful Web services return to the original design principles of the World Wide Web, and its REST style [10]. In this model, information and computation are abstracted as *resources*, which are manipulated using a fixed set of four CRUD (create, read, update, delete) operations. Since resources are central in this model, we say that the model is resource-oriented. In a context analogous to databases, CRUD languages for RESTful Web services have been developed as variants of the SQL language: see for instance the language YQL from Yahoo. But, contrary to the process-oriented model, there is no standard like BPEL, which has led to the current diversity of CRUD languages in use.

Because of the absence not only of a unified model for service-oriented computing, but also of a standard for CRUD languages, there is no universal language for manipulating both services and resources, which leads to some major issues, namely *adaptation*, *integration* and *coordination* problems. Let us illustrate these problems with two well-known Web photos management systems, Picasa and Flickr. Both provide CRUD interfaces for client applications. However, their resource models and CRUD interfaces differ. Hence, an *adaptation* is needed when a client application that communicates with Picasa must change to communicate instead with Flickr. An *integration* is needed when the client application must communicate with both Picasa and Flickr. A *coordination* is needed when two scripts, possibly written in distinct languages, must cooperate to manipulate resources managed by one service.

In this paper, we solve these problems in the simplest model, the resource-oriented one. We propose a pivot architecture built around a universal language for manipulating resources. The pivot architecture decreases the coupling between CRUD languages and CRUD interfaces, leading to a solution to the three problems mentioned above for the resource-oriented model. Central to the pivot architecture, the pivot language called CREOLE provides a universal, minimalist and formal way of defining CRUD scripts to manipulate resources.

The paper is organized as follows. First, after defining the problems of adaptation, integration and coordination, we introduce the pivot architecture and present related work. Second, we state the requirements for a universal language for manipulating resources and motivate its design, with respect to the state of the art. Then, we describe the language CREOLE, its syntax and its semantics, and validate its design against the requirements. Finally, we concretely show how the pivot architecture solves adaptation, integration and coordination problems in a paradigmatic use case, the management of photos in Flickr and Picasa. We conclude by a roadmap for future work. An important step is to extend our solution, to deal not only with the resource-oriented model, but also with the process-oriented model.

2 A pivot architecture

The absence of a unified service-oriented language for manipulating CRUD resources leads to several interoperability issues. Interoperability can be defined as the ability of two or more systems or

components to exchange information and to use the information that has been exchanged[1].Without interoperability, we are faced with composition problems related to adaptation, integration and coordination of heterogeneous services. By *adaptation*, we mean the problem of switching from one service provider to another without affecting its clients. By *integration*, we mean the problem of providing a unified interface for a set of resources managed by different CRUD interfaces. By *coordination*, we mean the problem of executing different scripts, possibly written in different languages, attempting to manipulate the same resources managed by one CRUD interface.

The pivot architecture described in Figure 1 solves these problems. It is built around a universal language for manipulating resources, called CREOLE (CRUD Language for Resource Edition). Scripts written in existing CRUD languages, like SQL, are compiled into the pivot language CREOLE and then executed over different CRUD interfaces, like Picasa's or Flickr's. To be effective, a pivot architecture relies on two assumptions. First, it must be possible to compile from any source language to the pivot language. We will briefly see that the language CREOLE satisfies this universality property with respect to CRUD languages. Thanks to this property, scripts written in different CRUD languages can be coordinated by using the Mediator design pattern [13, p. 273]. Second, it must be possible to interface the language CREOLE with the applications manipulating resources, characterized by their own resource representation and CRUD interface. We will see that the design of these interface connectors, called in the following built-in virtual machines, is akin to the design of RESTful Web services [22]. We also use other virtual machines, dedicated to the execution of the scripts written in the pivot language CREOLE. To resolve the adaptation and integration issues, the virtual machines are organized following two other design patterns, namely the Adapter and the Facade patterns [13, pp. 139, 185], for adaptation and integration respectively.

We identify several advantages of the pivot architecture. First, using a pivot language avoids the combinatorial explosion of translations, from multiple CRUD languages to different CRUD interfaces. Then, developers are allowed to program in their favorite CRUD language such as SQL or XQuery, with the additional advantage of being able to profit from the specific features offered by each language. Moreover, existing scripts written in different high-level languages can be executed on different CRUD interfaces without the need to be rewritten. Finally, the proposed pivot architecture overcomes the composition problems related to adaptation, integration and coordination.

[1] According to the IEEE Standard Computer Dictionary.

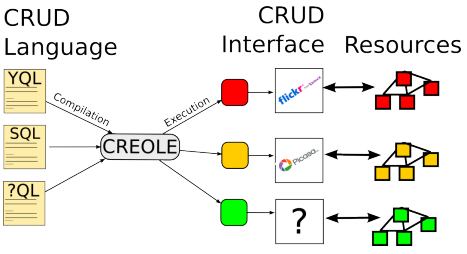

Figure 1: A Pivot Architecture

Related work In linguistics, a pivot language is an artificial or natural language used as an intermediary language for easing translation between many different languages (e.g. Interlingua, english). In computing, for analogous reasons, pivot infrastructures built around an intermediate language have been successful. For instance, virtual machines with their bytecode language are now common, allowing programs written in different languages to be compiled and executed over different architectures and systems (e.g. Java VM, .NET).

The pivot architecture can also benefit from techniques for the generation of mediators, adapters and facades. Instead of a manual generation as in Section 5, an automatic generation is possible, as exemplified by Brogi and Popescu [6] for BPEL processes, and by Mateescu, Poizat and Salaün for processes represented as symbolic transitions systems and also implemented in BPEL [21].

The main difficulty in a pivot architecture remains the design of the pivot language and its associated virtual machine. Various calculi, described in Bruni's comprehensive synthesis [7], have been proposed with the aim to capture aspects of service-oriented computing, from a verification or a modeling point of view but also from a formalization and programming point of view, which is related to our approach. However, these calculi are essentially process-oriented and not resource-oriented. As for the resource-oriented model, limited research have been undertaken in the formalization of RESTful Web services. Recently, Garrote and Moreno have proposed a language [19] combining a process calculus for the exchanges of messages and the coordination language LINDA [15] for the description of resource computations. Our solution presents the same two layers: a process language for distribution and a script language for resource computations, which as in Linda, includes operations for adding and deleting data in a shared dataspace, as we will see in the next sections.

3 Requirements and design rationale for the pivot language

First, we attempt to identify some essential requirements for the language CREOLE, considered as the language for editing resources at the heart of the pivot architecture. Second, we motivate the design with respect to the requirements.

Requirements The requirements can be split into two parts: general ones, relative to service-oriented computing, and particular ones, relative to the resource-oriented model.

Starting from the analysis led by Caires, Seco and Vieira [24, Sect. 2], and a general presentation of service-oriented computing [20], we have identified four general requirements: distribution, process delegation, scope management, and dynamic service binding. We do not deal with the requirements about distribution and process delegation, already well described by Caires et al. [24], but we focus on scope management and dynamic binding.

A client and a server execute in different contexts: entities used in the execution can be either local or shared between the server and the client. More interestingly, contexts dynamically evolve. For instance, a client can create a new session identifier that it sends to the server with its request. In its reply, the server also transmits the identifier that the client must use in order to relate the reply

to its request. Thus, name creation and name extrusion turn out to be two essential requirements. Name extrusion naturally leads to dynamic service binding, when the name represents a service, via its location. Dynamic binding is used for service discovery [20, Fig. 1] and dynamic routing, for instance in a well-known service interaction pattern [2] called Request with referral.

We have also identified requirements for the language CREOLE that are particular to the resource-oriented model: they deal with resource modeling and its consequences for a pivot language.

How to represent a resource? In the database field, since Codd's work, the data model has been defined as a relational model. Likewise, the markup language XML, used for representing data in web services, is founded on a relational model, as shown for instance by Benedikt and Koch's formalization of the query language XPATH [3]. We require that the language CREOLE adopts the relational model to represent resources, therefore assuming a logical approach. Following model theory, we represent resources as a structure, consisting of a universe and an interpretation over the universe of each relation in some signature, used to define the class of the resources considered.

Choosing the relational model results in two requirements for CREOLE, since a pivot language must satisfy two properties, universality and ability to interface, as seen in Section 2.

The relational model is equipped with natural operators, leading to the relational algebra: selection, projection, Cartesian product, set union, set difference, and renaming. We therefore require that the language CREOLE can express all these operations. More generally, we require that the language can express any computable transformation between structures: the language must be universal with respect to the relational model. For instance, it must be able to express aggregation and recursion, two powerful features, found natively but separately in SQL[2] and DATALOG[3] respectively.

In the relational model, a resource is represented as a relational structure. It has a uniform interface, namely a CRUD interface. A resource can be created or deleted by adding its complete representation to the structure or removing it respectively. It can be requested by querying the content of the structure and updated by modifying the structure. We therefore require that the CRUD interfaces of the relational structures can be mapped to the CRUD interfaces of the resources managed by the applications to which the language CREOLE is connected.

Design rationale Just as the requirements are split into two parts, the language CREOLE is designed with two layers, one defining scripts for resource manipulation and one defining processes for distribution.

Consistent with our logical point of view for representing resources, our script language is first inspired by DATALOG [8], a query language for deductive databases, in other words for structures in the relational model. However, DATALOG has a major limitation: it cannot express the deletion or the update of resources. Its semantics is essentially monotone: the representation of resources always increases during computations. Several disconnected lines of research have addressed this problem, for instance Zaniolo et al. have extended DATALOG with a notion of choice [18] or with

[2] See [16] for a formalization of SQL's semantics.
[3] See [8] for an introduction to DATALOG.

aggregate operators [25], and Ganzinger and McAllester [14] have allowed facts to be deleted and rules to be selected with priorities. Instead of using ad-hoc extensions, we choose to use linear logic as a foundation for our language. Two recent works have directly inspired our work.

First, Pfenning and Simmons have proposed a programming language in linear logic [23]. Besides persistent predicates, as found in DATALOG, there are ephemeral predicates, corresponding to linear resources. The operational semantics alternates a monotone deduction that involves only persistent predicates and a commitment corresponding to the firing of a rule consuming ephemeral atoms, which are propositions built from ephemeral predicates. Second, Betz, Raiser and Frühwirth have defined an extension based on linear logic for the language *Constraint handling Rules* [12] (CHR), a declarative language based on multiset rewriting, originally designed for writing constraint solvers and now employed as a general purpose language. They introduce persistent and ephemeral predicates [5] in order to ensure termination for so-called propagation rules, leading to a language akin to the preceding one.

Instead of using the distinction between persistent and ephemeral predicates, we use a distinction between relations and multi-relations. Multi-relations are multi-sets: an element in a multi-relation may have multiple occurrences. Relations are sets: an element in a relation has a unique occurrence. Exhaustive duplicate eliminations transform a multi-relation into a relation. This distinction leads to a more primitive mechanism. Indeed, whereas an ephemeral predicate is simply encoded as a multi-relation, a persistent predicate is encoded as a relation, and not a multi-relation, that satisfies an extra condition: all atoms built from a persistent predicate must be preserved by rules. Persistence can therefore be encoded.

Finally, generalizing the preceding languages based on linear logic, our script language is based on multiset rewriting. Thus, it has also its roots in the chemical reaction model: it can be considered as a variant of the language GAMMA [1]. More precisely, it is a restriction of a coordination language with schedulers [9] for a variant of GAMMA. Indeed, we have considered as linear resources not only the atoms but also the rules: rules are consumed when they are fired, except when they are replicable. There is also a sequence operator, allowing rules to be organized in distinct phases.

We now come to the distribution layer. Our process language is directly inspired by the join-calculus, a process calculus that can also be considered as a language for multiset rewriting, with a chemical semantics [11]. The join-calculus is interesting because of its natural notion of location and its implementability in a distributed setting. Rules are organized in definitions that are located. Given a channel, which is equivalent to our notion of predicate (multi-relation or relation), all the rules consuming atoms built from this channel belong to the same definition. Whenever an atom is generated, it is migrated to the unique definition dealing with the associated channel: this mechanism mimics a call from a client to the definition acting as a server. The join-calculus is also interesting because of its ability to express dynamic binding: indeed, channels can be communicated. Likewise, predicates can be communicated in CREOLE.

4 Design and validation of the pivot language CREOLE

We describe CREOLE syntax and its two layers, defining scripts for resource manipulation and processes for distribution, respectively. Table 1 sums up this syntax[4]. We then give the semantics of the language. We end the section by validating the design against the requirements.

Process	$p ::= (\overrightarrow{X})s \mid \texttt{let}\ \ p\ \ \texttt{in}\ \ p \mid p,p$	Script or Let server used in client or Parallel
Script	$s ::= \emptyset \mid r \mid s,s \mid s;s \mid s^{\omega}$	Skip or Rule or Parallel or Sequence or Replication
Rule	$r ::= j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2$	If j_1 then j_2 with new names $\overrightarrow{v}$ $\quad\left(\overrightarrow{v} = \mathrm{FV}(j_2) - \mathrm{FV}(j_1)\right)$
Molecule	$j ::= \emptyset \mid a \mid j \,\&\, j$	Conjunction of atoms
Predicate	$X ::= R \mid M$	Relation or Multi-relation
Atom	$a ::= X(\overrightarrow{X},\overrightarrow{v})$	Predicate applied to predicates and variables

Table 1: Language CREOLE – Scripts and processes

Scripts and processes The most primitive entities in CREOLE are predicates, either multi-relations or relations, and variables. Atoms are built using predicates and variables: a predicate X can be applied to a sequence $\overrightarrow{Y}$ of predicates, possibly empty, and to a sequence $\overrightarrow{v}$ of variables, giving atom $X(\overrightarrow{Y},\overrightarrow{v})$. Atoms $a_1,\ldots,a_p$ can be joined together to make a molecule $a_1 \,\&\, \ldots \,\&\, a_p$. The core part of CREOLE scripts are reactions that transform molecules into other molecules. A reaction is specified by a rule $j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2$, transforming any molecule matching the molecule pattern j_1 to a new molecule matching the molecule pattern j_2, using new variables in $\overrightarrow{v}$. A variable in j_2 is free if it occurs in j_2 without being declared in $\overrightarrow{v}$. In that case, it must be bound by the rule: it must occur in j_1. Finally, a CREOLE script can be seen as a specification of a schedule for rules. There are basic scripts, the empty one, which contains no rule and does nothing, and the singleton one, which contains a unique rule that can be fired only once. The parallel operator allows scripts to be concurrently active. For instance, the script r,r allows the rule r to be fired twice, whereas the script r,r' allows the rules r and r' to be fired exactly once each one, in any order. If a script needs to be executed an indefinite number of times, the replication operator can be used: for instance, the script r^{ω} means that the rule r is always ready to be fired. There is also a sequential operator, at any depth, allowing the transformations defined by scripts to be sequentially composed.

The distribution layer is defined around a process language: a process distributes scripts in a client-server architecture. The definition of a script is preceded with the declaration of the public predicates provided by the script. Two processes can be put in parallel: they execute concurrently without directly communicating. To enable a direct communication between two processes, the initial emitter or caller needs to be declared as a client, and the initial receiver or callee as a server. Consider the process $\texttt{let}\ \ p_{\mathrm{s}}\ \ \texttt{in}\ \ (D)s$, where p_{s} is the server process and $(D)s$ the client process,

[4] As usual, we denote by $\mathrm{FV}(t)$ the set of free variables occurring in the term t. The notation $\overrightarrow{x}$ denotes a sequence of x, when the particular members of the sequence do not matter; the sequence may be empty.

equal to a simple script s declaring public predicates in D. Each rule $j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2$ defined in script s can use in j_1 and j_2 the predicates in D. But it can also produce in j_2 atoms built from predicates declared as public in the server process p_s. In other words, a client can invoke a server. How does the server reply to the client? The client cannot directly consume atoms from predicates declared as public in the server process. Indeed, this interaction would violate the locality principle that we impose to the process language, in conformity with the join-calculus [11]: for each public predicate, there is one, and only one, script consuming this predicate, the script where the predicate is declared, which allows a very simple implementation for atom communication. Actually, the client must transmit to the server a reference to one of its own public predicate, which then can be used by the server to reply. Thus, we introduce second-order predicates, $X(\overrightarrow{Y}, \overrightarrow{v})$, which are applied to predicates $\overrightarrow{Y}$ and variables $\overrightarrow{v}$, in addition to first-order predicates, $X(\overrightarrow{v})$, only applied to variables $\overrightarrow{v}$. Thus, the rule $j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2$ can also produce in j_2 atoms built from predicates bound by j_1. A predicate used in the script that is neither public nor bound is private: it is not usable outside of the script.

Semantics with distributed chemical abstract machines The operational semantics of our script language is given by a reflexive chemical abstract machine [11]. Due to the lack of space, it is informally given in this paper, with some approximations. Its complete and accurate definition can be found in a technical report [17].

A configuration γ of the machine consists of two parts, $\rho \vdash \sigma$, where ρ is the reaction part, a multiset of executing scripts, and σ is the solution part, a multiset of molecules. There is a standard structural congruence between configurations, as described by Berry and Boudol for the π-calculus [4]. It expresses for the multiset union – denoted by a comma – associativity, commutativity and neutrality of the empty script and of the empty molecule – both denoted by $\emptyset$ –, and for the scope operator ν, the standard rules for name creation and extrusion. There are also two rules defining operators of the language:

Fusion and fission $\quad \rho \vdash \sigma, j_1 \,\&\, j_2 \equiv \rho \vdash \sigma, j_1, j_2$ $\qquad$ Replication $\quad \rho, s^\omega \vdash \sigma \equiv \rho, s^\omega, s \vdash \sigma$

Fission builds molecules from atoms whereas fusion is the reverse operation, which gives the meaning of the join operator &. As for the replication law, it gives the meaning of the replication operator: a replicated script is always available for execution.

The execution of a configuration is defined in three steps. First the *duplicate elimination* $\Rrightarrow$ eliminates every duplicated relational atom.

Duplicate elimination $\quad \rho \vdash \sigma, R(\overrightarrow{v}), R(\overrightarrow{v}) \Rrightarrow \rho \vdash \sigma, R(\overrightarrow{v})$

The duplicate elimination, with possible fusions to decompose molecules, is exhaustively performed between each reduction step to ensure that relational atoms occur at most once in a configuration. The *chemical reduction* $\rightarrow$ describes the basic reduction of the chemical abstract machine. There are two main rules.

Reaction $\quad \dfrac{}{\rho, (j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2) \vdash \sigma, j_1[\tau] \rightarrow \rho \vdash \sigma, (\overrightarrow{v}\overrightarrow{v}.j_2)[\tau]}$

The first rule deals with the main mechanism, reaction. The reaction rule $j_1 \triangleright \overrightarrow{v}\overrightarrow{v}.j_2$ is fireable when a molecule matches the molecule pattern j_1. The firing generates a new molecule matching the molecule pattern j_2, using new variables in $\overrightarrow{v}$; it consumes not only the molecule matching the molecule pattern j_1 but also the reaction rule.

$$\text{Sequence} \qquad \frac{\neg(\rho_1 \vdash \sigma \Rightarrow)}{\rho\,,(\rho_1\,;\rho_2) \vdash \sigma \rightarrow \rho\,,\rho_2 \vdash \sigma}$$

The second rule deals with the sequence operator. When the left part ρ_1 of the sequence script does not progress, it can be skipped. It remains to define *progression* $\Rightarrow$ from reduction. Assume that configuration γ_1 reduces to configuration γ_2: $\gamma_1 \rightarrow \gamma_2$. After an exhaustive duplicate elimination, configuration γ_2 becomes configuration γ_3. We say that the machine progresses from γ_1 to γ_3, denoted $\gamma_1 \Rightarrow \gamma_3$, if γ_1 is not structurally equivalent to γ_3, $\gamma_1 \not\equiv \gamma_3$. It means that either the reaction part, the solution part, or both, have changed.

Finally, the machine proceeds as follows. Starting from an initial configuration with no duplicates, it looks for a progression, possibly by using the fusion and fission rules and the replication rule. If no progression can happen, then the configuration is final. Otherwise, it non-deterministically chooses a possible progression, executes the associated reduction and exhaustively eliminates duplicates in the resulting configuration.

Now, we come to the semantics of the process language. Given a process, we associate to each script $(D^+)s$ declared in the process a chemical abstract machine, called a virtual machine, having as interface the public predicates declared in D^+. A distributed configuration δ contains two parts, a multiset of atoms a migrating between virtual machines and a set of local configurations $[\gamma_i]_{D_i}$, where for each virtual machine i associated to the process, γ_i is its local configuration and $D_i = D_i^+ \cup D_i^-$ the declaration of its predicates, either public (in D_i^+) or private (in D_i^-). The progression relation between distributed configurations is an extension of the progression relation defined for an individual virtual machine.

$$\text{Local} \qquad \frac{\gamma_i \Rightarrow \gamma_i'}{\delta, [\gamma_i]_{D_i} \Rightarrow \delta, [\gamma_i']_{D_i}}$$

It also contains two rules for the migration of atoms.

$$\text{Out} \quad \frac{a = X(\overrightarrow{Y}, \overrightarrow{v}) \quad X \notin D_i}{\delta, [\rho_i \vdash \sigma_i, a]_{D_i} \Rightarrow \delta, [\rho_i \vdash \sigma_i]_{D_i}, a} \qquad \text{In} \quad \frac{a = X(\overrightarrow{Y}, \overrightarrow{v}) \quad X \in D_i^+}{\delta, [\rho_i \vdash \sigma_i]_{D_i}, a \Rightarrow \delta, [\rho_i \vdash \sigma_i, a]_{D_i}}$$

After we have defined the syntax and the semantics of the language CREOLE, we now assess the design with respect to the requirements presented in Section 3.

Validation against requirements For validation, we consider the following requirements: distribution, with two aspects – implementation and expressivity –, scope management and dynamic service binding, script expressivity and ability to interface.

Thanks to its distributed semantics, implementing the language in a distributed context is easy. It suffices to assign to each script and its associated virtual machine a definite location

like a Uniform Resource Locator (URL). Then each atom built from a public predicate needs to convey the location where the predicate is declared, in order to allow the atom to be migrated when it is produced in another virtual machine. The only communication primitive in CREOLE is atom migration, corresponding to an asynchronous one-way invocation. As an atom can contain as argument a predicate for reply, dynamic binding is present for predicates, allowing different request-reply interactions, synchronous or not, to be encoded. For instance, let s be the following server script: $(I(K) \triangleright K())^{\omega}$. Then an echo interaction can be described as follows: `let` $(I)s$ `in` $(\emptyset \triangleright I(K)), (K() \triangleright \ldots)$.

Scope is statically managed for predicates, with the distinction between public and private predicates. Name creation is available for variables, and name extrusion for predicates and variables. Thus, a virtual machine can control its state and share relevant names. For instance, the precedent example can be refined in order to manage a session, allowing the reply to be related to the request: `let` $(I)(I(x,K) \triangleright K(x))^{\omega}$ `in` $(\emptyset \triangleright \nu x.I(x,K) \,\&\, W(x)), (W(x) \,\&\, K(x) \triangleright \ldots)$.

As for the expressivity requirements with respect to the relational model, it is easy to show that any operation in the relational algebra can be encoded in our script language. Aggregation can also be encoded. For instance, the script $\emptyset \triangleright C(0), \big(C(n) \,\&\, R(x) \triangleright C(n+1)\big)^{\omega}$ counts the number of elements in predicate R, assuming the availability of natural numbers, which can also be encoded in a straightforward manner. It is therefore possible to encode any SQL query in our script language, allowing the definition of a compiler. As for recursion, it is natively supported by our script language. For instance, DATALOG with negation, equipped with its well-founded semantics, can be encoded [17].

In the relational model, resources are represented as relational structures, using multi-relations when the number of occurrences matters, and using relations otherwise. All CRUD operations over relational structures can be mapped to HTTP operations over resource representations, precisely to `PUT`, `GET`, `POST` and `DELETE` respectively. This correspondence paves the way for an implementation with RESTful Web services of the built-in virtual machines, connecting the language CREOLE to the applications manipulating resources.

Thus, the language CREOLE satisfies the requirements that we have defined. In the next section, we illustrate the use of our pivot architecture and language in a paradigmatic use case.

5 Use Case: Photo Management on Flickr and Picasa

Flickr and Picasa are Yahoo's and Google's respective photo management systems. They offer web interfaces (APIs) to enable client applications to publish and organize photos on-line. These interfaces are essentially CRUD interfaces, implemented as RESTful web services and allowing photos to be manipulated as resources. This section illustrates the use of our pivot architecture and of CREOLE to solve adaptation, integration and coordination problems in the case of photo management with Flickr and Picasa. Concretely, our solution is based on three general design patterns, Adapter, Facade and Mediator.

Problem I: Adaptation Yahoo proposes a SQL-like language called YQL to query web services as if they were tables. In YQL, web services are represented as *virtual tables* wherein columns are mapped to input and output parameters. For example, the Flickr CRUD interface contains a method called `flickr.photo.counts` to count photos in a given date range. This service is represented in YQL as a virtual table called `PhotoCounts` with `fromDate` and `toDate` as input columns and `count` as an output column. The method can be called from a YQL query, akin to a SQL query:

```
SELECT count FROM PhotoCounts
WHERE fromDate ="01/01/2009" AND toDate="31/12/2009"
```

This script is mapped to a call to method `flickr.photo.counts`. Columns `fromDate` and `toDate` correspond to the method's input parameters, and column `count` to one of the output parameters.

How can we adapt the YQL script to count photos on Picasa, knowing that its CRUD Interface does not offer a count operation? Figure 2 summarizes our approach. In (a), the YQL script is compiled into CREOLE, then executed on a virtual machine (C-VM). In (b), we implement a virtual machine (A-VM), an Adapter allowing to switch from Flickr to Picasa. In this schema, virtual machines can be compared to components whose provided interfaces are relations, represented here by flat rectangles. We now detail the approach.

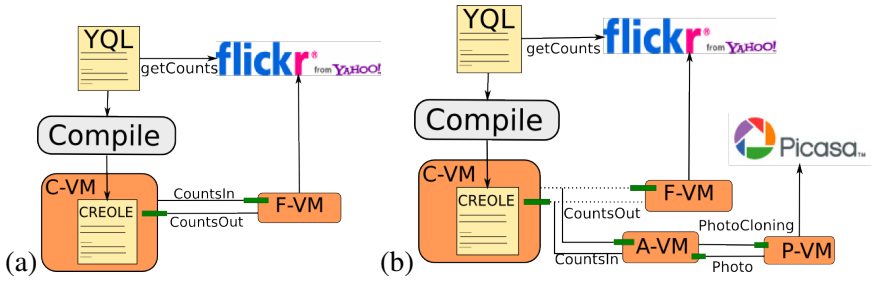

Figure 2: Flickr-Picasa Adaptation

To compile the YQL script into CREOLE, we map the virtual table `PhotoCounts` to relations `CountsIn` and `CountsOut`. Input columns `fromDate` and `toDate` are mapped to `CountsIn`'s parameters, and output column `count` is mapped to `CountsOut`'s last parameter. The following is the resulting CREOLE script:

1: $\emptyset \triangleright \nu x.\texttt{CountsIn}(x,\text{"01/01/2009"},\text{"31/12/2009"},\texttt{CountsOut}) \,\&\, \texttt{Session}(x),$
2: $\texttt{Session}(x) \,\&\, \texttt{CountsOut}(x,n) \triangleright \texttt{Result}(n)$

The fresh variable x, representing a session identifier, is used to relate the reply to the request. Note that the request transmits the relation `CountsOut` where it will obtain the reply.

The compiled YQL script is executed on a client virtual machine (`C-VM`). This virtual machine communicates with Flickr's built-in virtual machine (`F-VM`) which serves as a connector to Flickr's CRUD interface. Built-in virtual machines, like `F-VM`, are programmed to map CRUD operations over relations to RESTful Web services, accessed by HTTP requests.

In the second part of our solution, we create an adaptation virtual machine (A-VM) to adapt the desired behavior to Picasa's built-in virtual machine (P-VM). The following is the script executed in A-VM[5].

1: ($\texttt{CountsIn}(x, from, to, K) \triangleright \nu y.\texttt{Response}(x, y, from, to, 0, K) \,\&\, \texttt{PhotoCloning}(\texttt{Photo}, y)$,
2: ($\underline{\texttt{NotNull}}(id) \,\&\, \underline{\texttt{Between}}(from, date, to) \,\&\, \texttt{Response}(x, y, from, to, n, K) \,\&\, \texttt{Photo}(y, id, date) \triangleright$
$\texttt{PhotoCloning}(\texttt{Photo}, y) \,\&\, \texttt{Response}(x, y, from, to, n+1, K)$,
3: $\underline{\texttt{NotNull}}(id) \,\&\, \underline{\texttt{NotBetween}}(from, date, to) \,\&\, \underline{\texttt{Response}}(x, y, from, to, n, K) \,\&\, \texttt{Photo}(y, id, date) \triangleright$
$\texttt{PhotoCloning}(\texttt{Photo}, y)$ $)^{\omega}$,
4: $\underline{\texttt{Null}}(id) \,\&\, \texttt{Photo}(y, id, date) \,\&\, \texttt{Response}(x, y, from, to, n, K) \triangleright K(x, n)$ $)^{\omega}$

To count the photos taken between the two dates, the script uses P-VM's relation PhotoCloning. Given an identifier y, when the built-in virtual machine P-VM receives a request $\texttt{PhotoCloning}(K, y)$ for the first time, it produces a relation containing the relevant photos, by addressing HTTP GET requests to the Picasa server and answers by sending a first photo using the relation K. Then at each request $\texttt{PhotoCloning}(K, y)$, P-VM sends a new photo of the relation produced over K. When there is no more photo in the relation, it sends a photo with null as identifier. In the A-VM script, each time a photo is received, a request for another photo is sent (cf. lines 2 and 3); moreover, when the date of the photo satisfies the comparison criterion, the counter is incremented. When the null identifier is received, indicating that there are no more photos, the answer is sent to the client (cf. line 4). The whole script is replicated in order to indefinitely satisfy requests.

Finally, to switch from Flickr to Picasa, all we need to do is to change the virtual machine used as a server, from F-VM to A-VM, as follows: `let (CountsIn, ...) A-VM in C-VM`.

Problem II: Integration Despite the fact that both Picasa and Flickr manage similar resources, most of the time they are not represented in the same way. For instance, if photos in Picasa are represented by the relation $\texttt{Photo}(id, date, \overrightarrow{x})$, then in Flickr they are represented by the relation $\texttt{Photo}(id, date, \overrightarrow{y})$, where $\overrightarrow{x}$ and $\overrightarrow{y}$ do not have the same elements.

Nevertheless, CREOLE facilitates the implementation of an integration solution, like the one shown in Figure 3. In this schema, an intermediate virtual machine (I-VM), implementing a Facade, provides a common representation for photos in Flickr and Picasa, which is then used by the client virtual machine (C-VM).

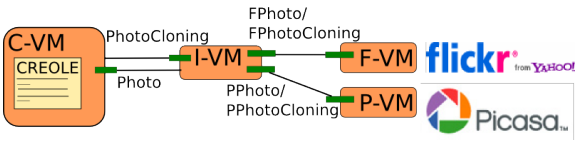

Figure 3: Flickr-Picasa Integration

[5]For readability, the rule $R \,\&\, S \triangleright R \,\&\, Q$ is written $\underline{R} \,\&\, S \triangleright Q$, where we have underlined the persistent atom R. We also use natural numbers, and the relations Between and NotBetween to compare dates, and Null and NotNull to test nullity, assuming their availability.

In this configuration, built-in virtual machines `F-VM` and `P-VM` provide both a relation to obtain photo information. As above, we call these relations `FPhotoCloning` and `PPhotoCloning`. Since the intermediate virtual machine (`I-VM`) holds a common representation for photos, it also provides a relation `PhotoCloning`, combining the attributes of Picasa's and Flickr's photos in the response. The following is the script corresponding to `I-VM`.

$$
\begin{array}{ll}
1: & \Big(\quad \texttt{PhotoClonning}(P,x) \triangleright \texttt{PPhotoCloning}(\texttt{PPhoto},x) \,\&\, \texttt{Response}(P,x), \\
2: & \quad \Big(\quad \underline{\texttt{NotNull}}(id) \,\&\, \texttt{PPhoto}(x,id,date,\overrightarrow{p}) \,\&\, \underline{\texttt{Response}}(P,x) \triangleright \\
 & \qquad P(x,id,date,\overrightarrow{p'}) \,\&\, \texttt{PPhotoCloning}(\texttt{PPhoto},x), \\
3: & \qquad \underline{\texttt{Null}}(id) \,\&\, \texttt{PPhoto}(x,id,date,\overrightarrow{p}) \triangleright \texttt{FPhotoCloning}(\texttt{FPhoto},x), \\
4: & \qquad \underline{\texttt{NotNull}}(id) \,\&\, \texttt{FPhoto}(x,id,date,\overrightarrow{f}) \,\&\, \underline{\texttt{Response}}(P,x) \triangleright \\
 & \qquad P(x,id,date,\overrightarrow{f'}) \,\&\, \texttt{FPhotoCloning}(\texttt{FPhoto},x) \quad \Big)^{\omega}, \\
5: & \qquad \underline{\texttt{Null}}(id) \,\&\, \texttt{FPhoto}(x,id,date,\overrightarrow{f}) \,\&\, \texttt{Response}(P,x) \triangleright P(x,id,date,\overrightarrow{f'}) \quad \Big)^{\omega}
\end{array}
$$

The lists $\overrightarrow{p'}$ and $\overrightarrow{f'}$ contain the same attributes and are computed from some combination, between intersection and union, of attributes in $\overrightarrow{p}$ and $\overrightarrow{f}$ respectively. As a consequence, we can simultaneously execute queries of photos on both CRUD interfaces. For example, we could execute the script of the adaptation scenario to count all our photos on both Flickr and Picasa, by setting `I-VM` as the server instead of `P-VM`: `let (PhotoCloning, ...) I-VM in A-VM`.

Due to the lack of space, we have presented a simple scenario; nevertheless, there are more complicated differences between Flickr and Picasa that can be tackled with our approach. Consider, for instance, how photos are organized in both services: in Flickr, photos can be organized in sets but can also be on their own; in Picasa however, photos must belong to one and only one album. We can solve this problem by using a common representation for albums and sets, and then applying the same integration schema as in the example shown here.

Problem III: Coordination One of YQL's limitations is the lack of support for aggregation. With CREOLE it is possible to coordinate scripts written in different languages to take advantage of features provided by each language. Hence, we can combine YQL capacity for querying services as tables with SQL support for aggregation. In the example shown in Figure 4, a YQL script to select photos taken between 01/01/2009 and 31/12/2009 is coordinated with a SQL script that counts rows from a given relation. Here are the corresponding YQL and SQL queries:

```
SELECT * FROM PhotoSearch          | SELECT COUNT(*) FROM R
WHERE min_taken_date="01/01/2009"  |
AND max_taken_date="31/12/2009"    |
```

The virtual table `PhotoSearch` is a representation of Flickr's method `flickr.photo.search` which takes `min_taken_date` and `max_taken_date` as input parameters. Note in the SQL query that `R` can be any relation since the query is not bound to a concrete database implementation.

The YQL and SQL queries are compiled into CREOLE and executed on a coordination virtual machine (`C-VM`). The `C-VM` virtual machine uses the relations `PhotoSearch` and `SearchResult` provided by the built-in virtual machine `F-VM`. Given an identifier x, when `F-VM` receives a request $\texttt{PhotoSearch}(a,b,x)$, it produces a relation associated to x and containing the photos taken

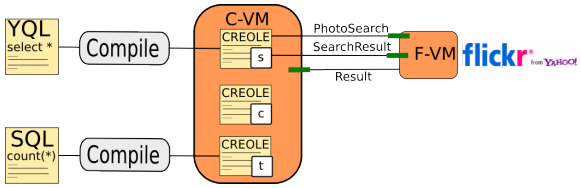

Figure 4: Coordination

between a and b, by addressing HTTP `GET` requests to the Flickr server. Then, at each request $\texttt{SearchResult}(K,x)$, `P-VM` sends a new photo of the relation produced over K. The following script s is the YQL query compiled into CREOLE:

1: $\emptyset \triangleright \nu\, x.\texttt{PhotoSearch}(\text{"01/01/2009"},\text{"31/12/2009"},x)\,\&\,\texttt{SearchResult}(\texttt{Result},x)$,
2: $\left(\underline{\texttt{NotNull}}(id)\,\&\,\texttt{Result}(x,id,\overrightarrow{y}) \triangleright \texttt{SearchResult}(\texttt{Result},x)\,\&\,\texttt{Photo}(x,id,\overrightarrow{y})\right)^{\omega}$,
3: $\underline{\texttt{Null}}(id)\,\&\,\texttt{Result}(x,id,\overrightarrow{y}) \triangleright \emptyset$

The script s initiates the search by calling the server `F-VM`. Then, each time a photo is received, a request for another photo is sent (cf. line 2). Finally, when there is no more photo, the script ends. At the same time, the SQL query is compiled into the following script t:

$$\emptyset \triangleright \texttt{Count(0)}, \left(\texttt{Count}(n)\,\&\,R(\overrightarrow{y}) \triangleright \texttt{Count}(n+1)\right)^{\omega}$$

Finally, a third script c coordinates the previous scripts, implementing a Mediator:

$$\left(\texttt{Photo}(\overrightarrow{y}) \triangleright \texttt{R}(\overrightarrow{y})\right)^{\omega}$$

This script, in parallel with s and t, combines the outcomes of the YQL script s and the counting of the SQL script t with a renaming from `Photo` to `R`. It finally produce an atom $\texttt{Count}(n)$, where n is the number of photos.

6 Conclusion and Future Work

In the context of Service-Oriented Computing, we have identified three main problems related to service composition, namely *adaptation*, *integration* and *coordination*, due to the absence of a unified model for manipulating resources. We have presented our approach to tackle these problems, consisting of a pivot architecture, where existing languages for manipulating resources are compiled into a pivot language, called CREOLE, and then executed over different resource interfaces, which are CRUD interfaces. We have mainly introduced CREOLE, a universal language for resource manipulation, which is at the heart of our solution. The motivating example of photo management on services like Flickr and Picasa has concretely shown how our proposed architecture solves adaptation, integration and coordination problems, and how CREOLE can be used either as a CRUD language or as a target language for the compilation from existing CRUD languages.

Yet we have only explored the resource-oriented model for services. An extension towards the process-oriented model would be valuable: indeed, it will bring a unified foundation for service-oriented computing. Actually, the two models share a lot of similarity, since they follow a same

architecture with three layers. First, there are resources. Second, there are services, limited to CRUD operations for the resource-oriented model and extended to any computation for the process-oriented model. Third, there are processes or scripts for orchestrating services.

Our future work has therefore two main objectives. First, we want to develop the formal foundations of the language CREOLE, as begun in our technical report [17]. The main questions here are the development of the theory of the language, from operational semantics to bisimilarity, and the assessment of its expressive power. Second, we want to implement the language and the whole pivot architecture. Four questions are here important: implementation of the chemical abstract machine and of its distribution, design of compilers into CREOLE for existing languages like YQL and BPEL, design of a user-friendly programming language based on the core calculus presented here, implementation of built-in virtual machines by connecting to RESTful services and WS-* services. Thus, these objectives pave the way to a unified foundation for service-oriented computing, in a theoretical and practical perspective.

Acknowledgments We are grateful to the anonymous reviewers of FOCLASA 2010 for their useful suggestions to improve this paper.

References

[1] Jean-Pierre Banâtre & Daniel Le Métayer (1990): *The GAMMA Model and Its Discipline of Programming*. *Science of Computer Programing* 15(1), pp. 55–77.

[2] Alistair Barros, Marlon Dumas & Arthur ter Hofstede (2005): *Service Interaction Patterns*. In: *Business Process Management, 3rd International Conference, BPM 2005*, *LNCS* 3649, Springer-Verlag, pp. 302–318.

[3] Michael Benedikt & Christoph Koch (2008): *XPath Leashed*. *ACM Computing Surveys* 41(1), pp. 1–54.

[4] Gérard Berry & Gérard Boudol (1992): *The Chemical Abstract Machine*. *Theoretical Computer Science* 96(1), pp. 217–248.

[5] Hariolf Betz, Frank Raiser & Thom Frühwirth (2010): *A Complete and Terminating Execution Model for Constraint Handling Rules*. In: *Logic Programming, 26th International Conference, ICLP '10*.

[6] Antonio Brogi & Razvan Popescu (2006): *Automated Generation of BPEL Adapters*. In: *Service-Oriented Computing - ICSOC 2006, 4th International Conference, Proceedings*, *LNCS* 4294, Springer-Verlag, pp. 27–39.

[7] Roberto Bruni (2009): *Calculi for Service-Oriented Computing*. In: *Formal Methods for Web Services, 9th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2009, Advanced Lectures*, *LNCS* 5569, Springer-Verlag, pp. 1–41.

[8] Stefano Ceri, Georg Gottlob & Letizia Tanca (1989): *What You Always Wanted to Know About Datalog (And Never Dared to Ask)*. *IEEE Transactions on Knowledge and Data Engineering* 1, pp. 146–166.

[9] Michel Chaudron & Edwin de Jong (1996): *Towards a Compositional Method for Coordinating Gamma Programs*. In: *Coordination Languages and Models, First International Conference, COORDINATION 1996, Proceedings*, *LNCS* 1061, Springer-Verlag, pp. 107–123.

[10] Roy Thomas Fielding (2000): *Architectural styles and the design of network-based software architectures*. Ph.D. thesis, University of California, Irvine.

[11] Cédric Fournet & Georges Gonthier (1996): *The reflexive CHAM and the join-calculus*. In: *Proceedings of the 23th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL '96)*, ACM Press, pp. 372–385.

[12] Thom Frühwirth (2008): *Welcome to Constraint Handling Rules*. In: Tom Schrijvers & Thom Frühwirth, editors: *Constraint Handling Rules*, *LNCS* 5388, Springer, pp. 1–15.

[13] Erich Gamma, Richard Helm, Ralph Johnson & John Vlissides (1995): *Design Patterns*. Addison-Wesley, Boston, MA.

[14] Harald Ganzinger & David McAllester (2002): *Logical Algorithms*. In: *Logic Programming, 18th International Conference, ICLP 2002, Proceedings*, *LNCS* 2401, Springer-Verlag, pp. 209–223.

[15] David Gelernter (1985): *Generative Communication in Linda*. *ACM Transactions on Programming Languages and Systems* 7(1), pp. 80–112.

[16] Martin Gogolla (1994): *Formal semantics of SQL*. In: *An Extended Entity-Relationship Model - Fundamentals and Pragmatics*, *LNCS* 767, Springer-Verlag, pp. 99–120.

[17] Hervé Grall & Nicolas Tabareau (2010): *Linear logic as a foundation for service-oriented computing*. Work in progress, available on HAL (`http://hal.archives-ouvertes.fr`), EMN-INRIA.

[18] Sergio Greco & Carlo Zaniolo (2001): *Greedy Algorithms in Datalog*. *Theory and Practice of Logic Programming* 1(4), pp. 381–407.

[19] Antonio Garrote Hernández & María Moreno García (2010): *A Formal Definition of RESTful Semantic Web Services*. In: *First International Workshop on RESTful Design (WS-REST 2010)*, pp. 39–45.

[20] Michael Huhns & Munindar Singh (2005): *Service-Oriented Computing: Key Concepts and Principles*. *IEEE Internet Computing* 9(1), pp. 75–81.

[21] Radu Mateescu, Pascal Poizat & Gwen Salaün (2008): *Adaptation of Service Protocols Using Process Algebra and On-the-Fly Reduction Techniques*. In: *Service-Oriented Computing - ICSOC 2008, 6th International Conference, Proceedings*, *LNCS* 5364, Springer-Verlag, pp. 84–99.

[22] Cesare Pautasso, Olaf Zimmermann & Frank Leymann (2008): *Restful web services vs. "big"' web services: making the right architectural decision*. In: *Proceedings of the 17th International World Wide Web Conference (WWW 2008)*, pp. 805–814.

[23] Robert Simmons & Frank Pfenning (2008): *Linear Logical Algorithms*. In: *Automata, Languages and Programming, Proceedings of the 35th International Colloquium, ICALP 2008*, *LNCS* 5126, Springer-Verlag, pp. 336–347.

[24] Hugo Vieira, Luís Caires & João Seco (2008): *The Conversation Calculus: a Model of Service Oriented Computation*. In: *17th European Symposium on Programming, ESOP 2008*, *LNCS* 4960, Springer-Verlag, pp. 269–283.

[25] Haixun Wang & Carlo Zaniolo (2000): *User-Defined Aggregates in Database Languages*. In: *7th International Workshop on Database Programming Languages, DBPL'99*, *LNCS* 1949, Springer-Verlag, pp. 43–60.