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

Mayleen Lacouture, Hervé Grall, Thomas Ledoux

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 [21].

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 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 been brought forward: RESTful web services return to the original design principles of the World Wide Web, and its REST style [10]. In this model, any information that can be named is abstracted as a *resource* and resources are manipulated using a fixed

Submitted to:
FOCLASA 2010

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.

The current situation is therefore characterized by the absence of a unified model for manipulating 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 services propose APIs for client applications. However, they provide each their own resource model and CRUD interface. In this context, an *adaptation* is needed when a client application, which communicates with the Picasa CRUD interface must change to communicate instead with the Flickr CRUD interface. An *integration* is needed when the client application must communicate with both Picasa and Flickr CRUD interfaces. A *coordination* is needed when two scripts, possibly written in distinct languages, must cooperate to manipulate resources managed by one service.

In this paper, to address these problems, we propose a pivot architecture built around a universal language to manipulate resources. The pivot architecture decreases the coupling between CRUD languages and CRUD interfaces, leading to a solution to the three problems mentioned above. Central to the pivot architecture, the pivot language called CREOLE provides a universal, minimalist, well-designed 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. Second, after stating the requirements for a universal language for manipulating resources, we formally describe the language CREOLE and informally motivate its definition with respect to the requirements. For each proposal, the architecture and the language, we compare with related work in the corresponding sections. Third, we concretely show how the pivot architecture solves adaptation, integration and coordination problems in the case of photo management in Flickr and Picasa. We conclude by a roadmap for future work.

2 A Pivot Architecture

The absence of a unified model for manipulating CRUD resources in the context of SOC development 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 processes, probably written in different languages, attempting to manipulate the same resources managed by one CRUD interface.

To address these issues, we propose a pivot architecture (see Figure 1) built around a universal language to manipulate resources, called CREOLE (CRUD Language for Resource Edition). Scripts written in existing CRUD languages, like SQL, are compiled into the pivot language CREOLE and

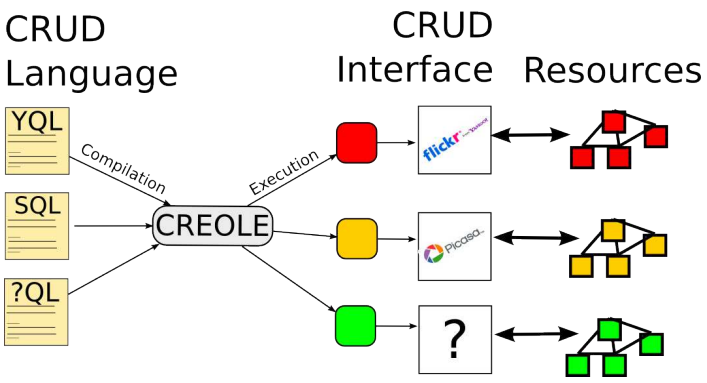

Figure 1: A Pivot Architecture

then executed over different CRUD interfaces, like Picasa's or Flickr's. The compilation of CRUD languages towards the same pivot model allows us to coordinate scripts written in different languages. Since CREOLE language generalizes query languages (see the next section), the compilation is always possible. Then, since each web application provides its own resource model, resource representations and CRUD interface, it is necessary to develop some connectors to allow interactions between CREOLE and a web application. The design of these connectors will be akin to the design of web services, thus inheriting from their implementation neutrality [19]. In the following, a connector will correspond to built-in virtual machines. Other virtual machines are present, dedicated to the execution of the scripts written in the pivot language CREOLE. To connect these virtual machines, we will resort to several design patterns [13]. More precisely, we use the Adapter and the Facade patterns to resolve the adaptation and integration issues respectively.

We identify several advantages of the pivot architecture/language design. 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. Finally, existing scripts written in different high-level languages can be executed on different CRUD interfaces without the need to be rewritten.

To conclude, the proposed architecture based on a pivot model answers well to the composition problems related to adaptation, integration and coordination of services. The main difficulty is the design of the pivot language itself to propose a universal, minimalist and formal way of defining CRUD scripts to manipulate resources.

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 writing in different languages to be compiled and executed over different architectures and systems (e.g. Java VM, .NET). The main question in a pivot architecture is the design of the pivot language

and its associated virtual machine.

Various calculi [7, 20, 23][1] 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 [2] 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.

3 A CRUD Language for Resource Edition: from requirements to definition

Starting from requirements, and the design rationale, we define the language and finally give some paradigmatic examples.

3.1 Requirements

In this section, 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. These requirements can be split in two parts: general ones, dealing with service-oriented computing, and particular ones, dealing with the resource-oriented model.

Starting from the "attempt to isolate and clarify essential characteristics of the service-oriented model of computation", led by Caires, Seco and Vieira [23], and a general presentation of service-oriented computing, as found for instance in Huhns and Singh's article [19], we have identified four general requirements: distribution, process delegation, scope management, and dynamic service binding. We do not deal with distribution and process delegation, already described [23], and concentrate on scope management and dynamic binding.

A client and a server execute in different contexts: entities used in the execution can be either local, that is restricted to the client or to the server, or shared between the server and the client. More interestingly, contexts dynamically evolve. For instance, after a first request, the server can create a new session identifier. Then, in its reply, it transmits the identifier that the client must reuse in order to relate its subsequent requests to the first one. 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 [19, Fig. 1] and dynamic routing, for instance in a well-known service interaction pattern [4] called request with referral.

We have also identified three main requirements for the language CREOLE that are relevant to the resource-oriented model: data modeling, interface definition and expressivity.

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

[1] See the introduction in [23] for further references.

in web services, is founded on a relational model, as shown for instance by Benedikt and Koch's formalization of the query language XPATH [5]. Therefore, we require that the language CREOLE resorts to the relational model for representing resources. In other words, we adopt a logical point of view. 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 implies that resources have 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. Choosing the relational model also results in expressivity requirements. Indeed, 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. In other words, we require that the language is *universal* with respect to the relational model. Concretely, we will use two tests to evaluate its universality: the ability to express aggregation and the ability to express recursion. Indeed, beyond the relational operations, aggregation and recursion are two powerful features, found natively but separately in SQL[2] and DATALOG[3] respectively. Their combination leads to a significant extension, as advocated and shown for instance by Wang and Zaniolo in their proposal [24].

3.2 Design rationale

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

Since we have adopted a logical point of view for representing resources, our script language is first influenced by DATALOG [8], a query language for deductive databases, that is 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. that have extended DATALOG with a notion of choice [18] or with aggregate operators [24], and Ganzinger and McAllester [14] that allow 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 influenced our choice.

First, Pfenning and Simmons have proposed a programming language in linear logic [22]. Besides persistent relations, as found in DATALOG, there are ephemeral relations, corresponding to linear resources. The operational semantics alternates a monotone deduction that involves only persistent relations and a commitment corresponding to the firing of a rule that involves consuming ephemeral atoms, that is atomic propositions built from ephemeral relations.

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.

[2] See [16] for a formalization of SQL's semantics.

[3] See [8] for an introduction to DATALOG.

They introduce persistent and ephemeral relations [6] 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 relations, we use a distinction between relations and multi-relations. Multi-relations are multi-sets: an element in a multi-relation may have multiple occurrences. On the contrary, 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 relation is simply a multi-relation, a persistent relation is a relation, and not a multi-relation, that satisfies an extra condition: all atoms built from a persistent relation must be preserved by rules. Persistence is therefore under control.

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 [3]. 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. With this design, the script language satisfies the expressivity requirements, as we will shortly see in Section 3.4.

We now come to the distribution layer. Our process language is directly influenced 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 multi-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. The main differences between our language and the join-calculus are the absence of the linearity constraint on messages, which is needed to express common rules without a complicated encoding, and the distinction between channels and messages, which leads to a multi-sorted language with second-order relations, first-order relations and individuals.

At this stage, we can assert that a language conforming to the design choices that we have just described should satisfy all the requirements. However, we have not mentioned scope management. In the script language, as in CHR, it is possible to generate new variables representing individuals. In the process language, it is possible to define private relations in addition to the public relations that clients can call.

3.3 The language CREOLE

We now describe CREOLE's syntax and semantics, and its two layers, defining scripts for resource manipulation and processes for distribution respectively..

Resource manipulation The most primitive entities in CREOLE are *multi-relations*, *relations* and *variables*. Whereas variables are just names, relations have also an arity, a natural number.

From multi-relations, relations and variables, atoms are built, by applying any relation R or multi-relation M with arity n to a sequence of n variables, $v_1, \ldots, v_n$, to get atoms $R(v_1, \ldots, v_n)$ and $M(v_1, \ldots, v_n)$ respectively. Atoms $a_1, \ldots a_p$ can be joined together to make a molecule $a_1 \& \ldots \& a_p$.

The core part of CREOLE's scripts are reactions that transform molecules into other molecules. A reaction is specified by a rule $j_1 \triangleright \overrightarrow{v}.j_2$, transforming any molecule matching the molecule pattern j_1 to a new molecule matching the molecule pattern j_2, using new variables in $\overrightarrow{v}$[4]. Any variable free in j_2, that is occurring in j_2 without being declared in $\overrightarrow{v}$, 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, allowing the transformations defined by scripts to be sequentially composed. Table 1 sums up the syntax. As usual, we denote by $\mathrm{FV}(j)$ the set of free variables occurring in the molecule j.

The operational semantics of our language is given by a reflexive chemical abstract machine [11]. In this paper, it is informally given, with some approximations. Its complete and accurate definition can be found in a technical report [17], which is a work in progress dedicated to the theoretical foundations of the language. A configuration γ of the machine consists of two parts, $\rho \vdash \sigma$, where ρ is the reaction part, corresponding to the executing script, and σ is the solution part, a multiset of molecules. There is a standard structural congruence between configurations. The two main rules are the following:

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

Fission builds molecules from atoms whereas fusion is the reverse operation. 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	$\rho \vdash \sigma, R(\overrightarrow{v}), R(\overrightarrow{v}) \Rrightarrow \rho \vdash \sigma, R(\overrightarrow{v})$

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

The first rule deals with the main mechanism, reaction. The reaction rule $j_1 \triangleright \overrightarrow{v}.j_2$ is fireable when a molecule matches the molecule pattern j_1. The firing generates a new molecule matching

[4] Here, and in the following, the notation $\overrightarrow{x}$ denotes a sequence of x, or depending on the context, a set or multiset of x, when the particular members of the collection do not matter; the sequence, set or multiset may be empty.

Script	$s ::= \emptyset$	Skip
	$\mid r$	Rule
	$\mid s,s$	Parallel
	$\mid s;s$	Sequence
	$\mid s^{\omega}$	Replication
Rule	$r ::= j_1 \triangleright \overrightarrow{v}\,\overrightarrow{v}.j_2$	Transformation of j_1 into j_2 with new names $\overrightarrow{v}$ $(\overrightarrow{v} = \mathrm{FV}(j_2) - \mathrm{FV}(j_1))$
Molecule	$j ::= \emptyset \mid a \mid j \,\&\, j$	Conjunction of atoms
Atom	$a ::= R(\overrightarrow{v}) \mid M(\overrightarrow{v})$	Relation or multi-relation applied to variables

Table 1: Language CREOLE – Scripts

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 becomes inactive, that is 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 $\gamma_1 \Rightarrow \gamma_2$, if γ_1 is not structurally equivalent to γ_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. 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.

Distribution The distribution layer is defined around a process language. The distributed execution is organized from virtual machines, which are chemical abstract machines executing scripts. A virtual machine acts as a server or as a client, switching between the two roles: it can call a server or reply to a client. The process language allows the scripts associated to virtual machines to be defined. The definition of a script is preceded with the declaration of the relations[5] used in the script. Relations are either private, that is local to the script, or public, that is visible outside the script, which is then a server. A private relation can only be used in the script whereas a public relation can be used in another scripts, which are then clients. More precisely, the rules defined in a client script can produce atoms built from relations declared as public in a server script. In other words, a client can invoke a server. How does the server reply to the client? The client cannot directly consume atoms from relations declared as public in a server: indeed, this interaction would violate the locality principle that we impose to the process language, in conformity with the join-calculus [11]. Actually, the client must transmit to the server a reference to one of its own public relation, which then can

[5] We mean here and in the following relations and multi-relations.

be used by the server to reply. Thus, we introduce second-order relations, which are applied to first-order relations and variables, in addition to the first-order relations. Table 2 describes the full language, consisting of the process language and the script language.

3.4 Some simple examples

Consider a unary relation or multi-relation P. To define the initial state of P, we can use a simple rule:

$$\emptyset \triangleright \nu\, a.\nu\, b.P(a) \,\&\, P(b)\,;$$

After the unique firing of the rule, the relation or multi-relation is initialized: the solution contains two elements, a and b, two new variables. The rule can no longer be fired: actually, it is removed from the active script, which we have called the *reaction*.

To rename P as Q, we can then write:

$$(P(x) \triangleright Q(x))^{\omega}$$

We need to resort to the replication operator in order to allow multiple firings of the rule. Very often, rules are replicated. The rules that are not replicated are used to define solutions in particular states, or transitions between states. Note that without a replication operator, if all rules were implicitly replicated, it would be impossible to define an initial solution, as above, for termination reasons. The preceding rule applied twice consumes $P(a)$ and $P(b)$ and produces $Q(a)$ and $Q(b)$. If P and Q are both relations, or are both multi-relations, the script just describes a renaming operation. It is more interesting when P is a multi-relation and Q a relation. Assume that the solution initially contains atoms $P(a), P(a), P(b)$. After the conversion of the multi-relation P into a relation Q, the solution contains two atoms: $Q(a), Q(b)$. Indeed, duplicates are automatically eliminated for relations.
To make a clone of P, we can try:

$$(P(x) \triangleright P(x) \,\&\, Q(x))^{\omega}$$

The script works when Q is a relation. But when Q is a multi-relation, it diverges: the script generates a solution containing an infinity of atom $Q(a)$, from some atom $P(a)$ initially in the solution. Here is a right script for cloning:

$$(P(x) \triangleright P'(x) \,\&\, Q(x))^{\omega}\,;(P'(x) \triangleright P(x))^{\omega}$$

Without the sequence operator, it would be impossible to define a clone operation for multi-relations [17]. In other words, the sequence operator, like the replication operator, does increase the expressive power of our script language.

We now come to the expressivity requirements. We can easily encode not only primitive datatypes like booleans and integers[6], but also the relational algebra. For instance, the script $(\Delta(x) \triangleright P(x))^{\omega}$ adds the relation Δ to P whereas the script $(\Delta(x) \,\&\, P(x) \triangleright \emptyset)^{\omega}$ removes Δ from P. Recursion is native whereas aggregation is easy to program. For instance, to count the size of P, we proceed as follows: $\left(\emptyset \triangleright C(0)\,;(C(n) \,\&\, P(x) \triangleright C(n+1))^{\omega}\right)$.

[6] In the following, in some programming examples, we will directly use some datatypes to ease reading, with a slight abuse of notation since we will not resort to the encoding.

Process	p	$::= (D)s$	Script
		$\mid$ let p_{server} in p_{client}	Server defined in client
		$\mid p, p$	Parallel
Declaration	D	$::=$ public: $\overrightarrow{R_1}$; private: $\overrightarrow{R_2}$	Public and private relations
Script	s	$::= \ldots$	
Rule	r	$::= \ldots$	
Molecule	j	$::= \ldots$	
Atom	a	$::= R_2(\overrightarrow{R_1}, \overrightarrow{v})$	Second-order relation applied to first-order relations and variables
		$\mid R_1(\overrightarrow{v})$	First-order relation applied to variables

Table 2: Language CREOLE - Processes and scripts

As for the distributed side, a typical client-server interaction can be described as let s in c, where s is the server and c the client. For instance, an echo server is defined as follows:

$$s \stackrel{\text{def}}{=} (\text{public: } In; \text{ private: } \emptyset)\ In(K) \triangleright K()$$

A client will call the server with a rule $\emptyset \triangleright In(Out)$, passing the continuation, the second-order public relation Out, on which it will get the reply of the server, via a rule $Out() \triangleright \ldots$ If the server transmits some information in its reply, the client will probably need to relate the reply of the server to its request. The solution is to use a session identifier, with a call $\emptyset \triangleright \nu\, x.In(x, Out)$, passing a session identifier and the continuation. The reply $Out(x, \ldots)$ of the server will relate the session identifier x with the information transmitted.

4 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, 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.

4.1 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*, where columns map 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 as a virtual table called *PhotoCounts*, with input column *takenDates* and output columns *count, fromDate, toDate*. Then, the method can be called from a YQL query, akin to a SQL query:

```
SELECT count FROM PhotoCounts
WHERE takenDates ="01/01/2009,31/12/2009"
```

This script is mapped to a call to method *flickr.photo.counts*, where column *takenDates* corresponds to the method's input parameter, 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 only offers primitive REST operations? Figure 2 summarizes our approach: in (a) we begin by compiling the YQL script into CREOLE, which is then executed on a virtual machine (C-VM); in (b) we implement an adaptation virtual machine (A-VM) that allows us to switch from Flickr to Picasa. In this schema, virtual machines are comparable to components, whose provided interfaces are relations, represented by flat rectangles.

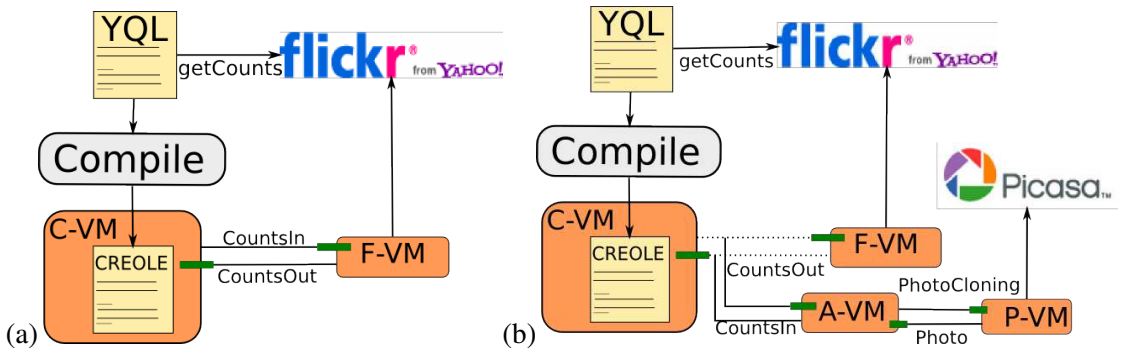

Figure 2: Flickr-Picasa Adaptation

The compiled YQL script is executed on a client virtual machine (C-VM). This virtual machine uses the relation *CountsIn* provided by Flickr's built-in virtual machine (F-VM), which directly communicates with Flickr's CRUD interface. To receive the result from *F-VM*, the client virtual machine provides as well the relation *CountsOut*. The result of compiling the YQL query into CREOLE is the following script:

$$\begin{array}{l|l} 1 & \emptyset \triangleright \nu\, x.CountsIn(x,\text{"01/01/2009"},\text{"31/12/2009"},CountsOut)\,\&\,Session(x) \\ 2 & Session(x)\,\&\,CountsOut(x,n) \triangleright Result(n) \end{array}$$

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*, used to obtain the reply. Next, we create an adaptation virtual machine (A-VM), which wraps Picasa's built-in virtual machine (P-VM). The adaptation virtual machine makes use of relation *PhotoCloning* on *P-VM*, which generate a copy of the internal photo relation. To obtain the result from *P-VM*, *A-VM* provides the relation *Photo*. The adaptation virtual machine also provides the required relation *CountsIn*. The following is the script corresponding to *A-VM*:

$$\begin{array}{l|l} 1 & \Big(CountsIn(x,from,to,K) \triangleright \nu\, y.Response(x,y,from,to,K,0)\,\&\,PhotoCloning(Photo,y); \\ 2 & \big(Date.Between(from,date,to)\,\&\,Response(x,y,from,to,K,n)\,\&\,Photo(y,id,date) \triangleright \\ & \quad Date.Between(from,date,to)\,\&\,Response(x,y,from,to,K,n+1)\big)^{\omega}; \\ 3 & Response(x,y,from,to,K,n) \triangleright K(x,n)\Big)^{\omega} \end{array}$$

Note that in this script we use the built-in relation $Date.Between(from,date,to)$ to calculate if $date$ occurs after date $from$ and before date to.

Finally, to switch from Flickr to Picasa, all we need to do is to change the virtual machine used as server, from *F-VM* to *A-VM*, as we do with in following specification:

$$let\ (public : CountsIn\ ...)\ A\text{-}VM\ in\ C\text{-}VM$$

4.2 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 $Photo(id, date, \overrightarrow{x})$, then in Flickr they are represented by the relation $Photo(id, date, \overrightarrow{y})$, 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), provides a common representation for photos in Flickr and Picasa, which is used by the client virtual machine (C-VM).

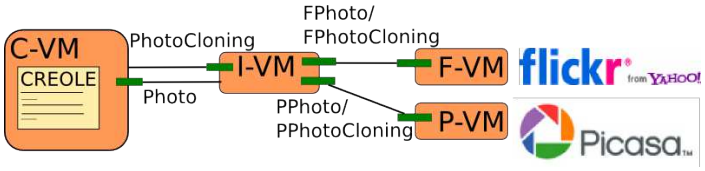

Figure 3: Flickr-Picasa Integration

In this configuration, built-in virtual machines *F-VM* and *P-VM* provide the same relation *PhotoCloning* as the one in the previous example. The intermediate virtual machine (I-VM) holds a common representation for photos. In fact *I-VM* provides also a relation *PhotoCloning*, except it combines the attributes of Picasa's and Flickr's photos in the response. The following is the script corresponding to *I-VM*:

$$\begin{array}{l|l} 1 & (PhotoClonning(x,P) \triangleright PPhotoCloning(PPhoto)\ \&\ FPhotoCloning(FPhoto)\ \&\ Response(x,P) \\ 2 & Response(x,P)\ \&\ PPhoto(id,date,\overrightarrow{p}) \triangleright P(x,id,date,\overrightarrow{p'}) \\ 3 & Response(x,P)\ \&\ FPhoto(id,date,\overrightarrow{f}) \triangleright P(x,id,date,\overrightarrow{f'})) \end{array}$$

The lists $\overrightarrow{p'}$ and $\overrightarrow{f'}$ are computed from $\overrightarrow{p}$ and $\overrightarrow{f}$ respectively. As consequence, we can simultaneously execute queries of photos on both CRUD interfaces. For example we could execute the script of Section 4.1, to count all our photos on both Flickr and Picasa, by setting *I-VM* as the server instead of *P-VM*:

$$let\ (public : PhotoCloning, ...)\ I\text{-}VM\ in\ A\text{-}VM$$

Due to the lack of space, we just 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 both services manage photo organization: in Flickr, photos can be organized in sets but can also be on their own; in Picasa however, photos must belong to one album and only one. We can solve this problem by using a common representation for albums and sets, and then applying the same integration schema from the example shown here.

4.3 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:

$$\begin{array}{l|l} SELECT * FROM\ PhotoSearch & SELECT\ COUNT(*)\ FROM\ R \\ WHERE\ min\_taken\_date = \text{"01/01/2009"} & \\ AND\ max\_taken\_date = \text{"31/12/2009"} & \end{array}$$

In the YQL query, 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.

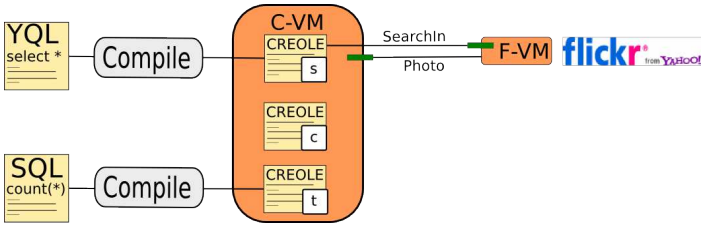

Figure 4: Coordination

The YQL and SQL queries are compiled into CREOLE and executed on a coordination virtual machine (*C-VM*). The *C-VM* virtual machine, where the scripts are executed, uses the relation *SearchIn* provided by Flickr's virtual machine (F-VM) and provides at the same time the relation *Photo*, to receive the response of the query. The following script *s* is the YQL query compiled into CREOLE:

$$\emptyset \triangleright SearchIn(\text{"01/01/2009"}, \text{"31/12/2009"}, Photo)$$

At the same time, the SQL query is compiled into the following script *t*:

$$\begin{array}{l|l} 1 & \emptyset \triangleright Count(0)\,; \\ 2 & \left(Count(n)\ \&\ R(\overrightarrow{y}) \triangleright Count(n+1)\right)^{\omega} \end{array}$$

Finally, a third script *c* is created to coordinate the previous scripts:

$$s, t, \left(Photo(\overrightarrow{y}) \triangleright R(\overrightarrow{y})\right)^{\omega}$$

This script combines the outcomes of the scripts corresponding to YQL and SQL with a renaming from *Photo* to *R*, to finally produce an atom $Count(n)$.

5 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 introduced CREOLE, a universal language for resource edition, which is at the heart of our solution, hence the importance we gave to it in this paper. The motivating example of photo management on services like Flickr and Picasa has 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, compilation of existing languages like BPEL or YQL into CREOLE, design of a real programming language instead of a core calculus as presented here, connexion 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.

References

[1] (1990): *IEEE Standard Computer Dictionary: A Compilation of IEEE Standard Computer Glossaries*. Institute of Electrical and Electronics Engineers, New York, NY.

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

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

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

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

[6] 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 (to appear)*.

[7] Michele Boreale, Roberto Bruni, Luís Caires, Rocco De Nicola, Ivan Lanese, Michele Loreti, Francisco Martins, Ugo Montanari, António Ravara, Davide Sangiorgi, Vasco Thudichum Vasconcelos & Gianluigi Zavattaro (2006): *SCC: A Service Centered Calculus*. In: *Web Services and Formal Methods, Third International Workshop, WS-FM 2006, Proceedings*, *Lecture Notes in Computer Science* 4184, Springer-Verlag, pp. 38–57.

[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, Lecture Notes in Computer Science 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, Lecture Notes in Computer Science 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, Lecture Notes in Computer Science 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, Lecture Notes in Computer Science 767, Springer-Verlag, pp. 99–120.

[17] Hervé Grall & Nicolas Tabareau (2010): *Linear logic as a foundation for service-oriented computing*. Work in progress, HAL.

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

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

[20] Ivan Lanese, Francisco Martins, Vasco Thudichum Vasconcelos & António Ravara (2007): *Disciplining Orchestration and Conversation in Service-Oriented Computing*. In: Fifth IEEE International Conference on Software Engineering and Formal Methods (SEFM 2007), IEEE Computer Society Press, pp. 305–314.

[21] 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.

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

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

[24] Haixun Wang & Carlo Zaniolo (2000): *User-Defined Aggregates in Database Languages*. In: Research Issues in Structured and Semistructured Database Programming, 7th International Workshop on Database Programming Languages, DBPL'99, Revised Papers, Lecture Notes in Computer Science 1949, Springer-Verlag, pp. 43–60.