2016 IEEE 30th International Conference on Advanced Information Networking and Applications

Data-driven Workflows for Microservices

Genericity in Jolie

Larisa Safina*, Manuel Mazzara*, Fabrizio Montesi†, Victor Rivera*
*Innopolis University, Russia
{l.safina, m.mazzara, v.rivera}@innopolis.ru
†University of Southern Denmark
fmontesi@imada.sdu.dk

***Abstract*—Microservices is an architectural style inspired by service-oriented computing that has recently started gaining popularity. Jolie is a programming language based on the microservices paradigm: the main building block of Jolie systems are services, in contrast to, e.g., functions or objects. The primitives offered by the Jolie language elicit many of the recurring patterns found in microservices, like load balancers and structured processes. However, Jolie still lacks some useful constructs for dealing with message types and data manipulation that are present in service-oriented computing. In this paper, we focus on the possibility of expressing choices at the level of data types, a feature well represented in standards for Web Services, e.g., WSDL. We extend Jolie to support such type choices, and enable Jolie processes to act on data generically (without knowing which type it has in the choice). We show the impact of our implementation on some of the typical scenarios found in microservice systems. This shows how computation can move from a process-driven to a data-driven approach, and leads to the preliminary identification of recurring communication patterns that can be shaped as design patterns.**

I. Introduction

The increasing complexity of modern software requires new approaches to architectural design and system modeling. Complex systems also show high level of concurrency, i.e. multiple intertwined threads of executions, often running on different hardware, which need to be synchronized and coordinated, and which much share information, often through different paradigms of communication. Therefore improving software quality and deploying reliable services is the consequence of an accurate use of optimal service-based architectural styles and well-established software engineering techniques for requirements elicitation, design, testing and verification, in particular when it comes to concurrent service-based systems. In order to tackle these issues from the architectural viewpoint, Microservices architecture appeared lately as a new paradigm for programming applications by means of the composition of small services, each running its own processes and communicating via light-weight mechanisms [8]. This approach has been built on the concepts of Service-oriented Architectures (SOA)[1] brought from crossing-boundaries workflows to the application level, and into the applications architectures, i.e. its Service-oriented Architecture and Programming from the large to the small.

Microservices architecture still shows distinctive characteristics which blend into something unique and different from SOA itself. The size is comparatively small versus a typical service [8], supporting the belief that the architectural design of a system is highly dependent on the structural design of the organization producing it.

In the context of microservices the Jolie programming language [13], [16] emerged as a paradigmatic solution tuned at getting the best out of this architectural style. Jolie initially began as an implementation of the SOCK process calculus [7], a formal model inspired by the CCS process calculus and the WS-BPEL programming language. Jolie is comprehensive and capable of implementing both simple services and complex orchestrations. Everything in Jolie is a microservice and all these microservices can be easily reused or composed for obtaining in turn new microservices. This approach supports distributed architecture and guarantees simple managing of components, which reduces maintenance and development costs.

The work presented in this paper is devoted to extending the Jolie programming language in order to support data-driven workflows, i.e. moving the control-flow decision-making from process-driven to data-driven. That means that control-flow can be directed at the time of message passing according to the nature of the message structure and type, a feature well represented in standards for Web Services (WSDL[2]), instead of requiring post-reception processing. A typical example of process-driven workflow is presented in [10]. In the microservice scenario this opportunity opens to novel programming patterns. Whether the process-driven or data-driven programming style is more suitable for an application really depends on the specific problem domain and, to some extent, to developers preferences and style. Data-driven flows have been realized in Jolie by means of an extension to its type system, which manifests as implementation of the choice type. Further extensions can be implemented in order to strengthen further this possibility. Currently, the Jolie type system enables users to define native, linked, and undefined types which lacks of expressiveness to add flexibility for users to implement, for instance, functions genericity or different behavior of the function based on arguments passed[1].

Two major contributions appear as a result of this work. The first is of scientific nature, i.e., moving from process-driven to data-driven and therefore open to new programming

[1]Here, by genericity, we do not mean type parametricity, but rather the possibility to define some functions/processes that handle data typed with choices without knowing exactly which choice has been made.

1550-445X/16 $31.00 © 2016 IEEE
DOI 10.1109/AINA.2016.95

patterns and styles. The second is of purely technical interest, and stands in the re-engineering of the Jolie interpreter as a consequence of the extended data type. This represents a relevant case study of interpreter re-engineering, and therefore can be valuable experience for the practitioners involved in activities of comparable complexity.

The paper is structured as follows: in Section II a quick overview of Jolie programming language is given in order to introduce the unfamiliar reader with the main concepts and syntax. Far from being an exhaustive report, the section is just a compendium providing the links for further study of the topic. Section III presents a case study on top of which the major narrative of the paper is built and the contributions are defined. In particular, two different kind of approaches computation are described: process-driven and data-driven, and examples of both are explained in order to understand the differences. Sections IV and V describe the architecture of the Jolie interpreter and the changes that were necessary in order to extend the type system. Finally, Section VI wraps up final considerations regarding the contribution of this work, and presents ideas on how future developments may be built up on top of current achievements.

II. The Jolie language

In this section we briefly recall the Jolie programming language in order to simplify the reading of the remaining part of the paper. This section explains both parts of a Jolie program: the deployment and behavioral parts. It also explains the communication of processes. Section II-D is devoted to explain in more details the Jolie's type system, which is part of the deployment part and is the main subject of this report.

A Jolie program can be formally expressed as:

`Program` ::= `Deployment Behavior`

A. Deployment part

The deployment part contains directives which help the Jolie program to receive and send messages and be orchestrated among other microservices. The deployment part is separated from the program behavioral part, so that the same behavior may be reused later with a different deployment configuration. Formally the `Deployment` part is expressed as:

`Deployment` ::= `DeploymentInstruction`$*$

Where `DeploymentInstruction` can include

- *Interfaces*: sets of operations equipped with information about their request (and sometimes response) types;
- *Message types*: can be represented as native types, linked types or undefined. Message types are the main subject of this report and shall be discussed later in section II-D.
- *Communication ports*: define how communications with other services are actually performed.

B. Behavioral part

The behavioral part contains microservice implementation of the functionalities, containing both computations and communication expressions. Examples of expanded behavioral part utilization will be provided later in section V-C. Formally, behavioral part can be expressed as:

```
Behavior          ::= BehaviouralBlock*
                      main {Process} BehaviouralBlock*

BehaviouralBlock  ::= define id {Process}
                  |   init {Process}
```

Where **`main`** is a procedure which defines an entry point of execution. It can be followed or preceded by define procedures with `id` identifier. **`init`** supports special procedures for initializing a service before it makes its behaviors available. Procedures specified with **`define`** can be used many times, while the one specified with **`init`** is executed only once, when the service is started.

`Process` defines the activities to be performed by the service. `Process`es can be composed in sequences, parallels and (input guarded) non-deterministic choices [13].

C. Communication

Communication of processes can be performed by two possible patterns: *one-way* (the endpoint receives a message) and *request-response* (the endpoint receives a message, and sends a response back to the caller):

```
Process           ::= ... | InputStatement
                  |   OutputStatement

InputStatement    ::= op(x)                 (One − Way)
                  |   op(x)(y) Process      (R − Response)

OutputStatement   ::= op@OPort(x)           (Notification)
                  |   op@OPort(x)(y)        (S − Response)
```

One-Way is used to receive a message for operation `op` in variable `x`. *R-Response* (Request-Response) is used to receive a message for operation `op` in variable `x`, execute a `Process` and then send back a response to the caller containing the value of variable `y`. *Notification* and *S-Response* (Solicit-Response) are the dual of the former ones to be used, respectively, for sending a message to a *One-Way* statement or to a *R-Response* one. `OPort` defines an output port name of desired endpoint.

D. Jolie type system

Jolie provides a language for describing the types that are allowed to be communicated over a network. Communications are type checked at run-time when a message is received [12], [14]. Message types are introduced in the deployment part of Jolie programs:

```
DeploymentInstruction  ::= ...
                       |   type id : TypeDefinition
```

Where `id` is an identifier in order to use the message type in other program parts and `TypeDefinition` can be a native type, native type with subtypes, native type with undefined subnodes, link type or can be undefined (means that variable is null until a value is assigned to it):

```
TypeDefinition  ::=  ...
                |    NativeType
                |    NativeType {SubTypeList}
                |    NativeType {?}
                |    id
                |    undefined

NativeType      ::=  int | double | string | raw | void | any
```

Where {?} represents untyped subnodes and `undefined` stands a shortcut for **any**:{?}.

Except commonly-used native types as **int**, **double** or **string**, Jolie also has the following types:

- **raw**, used for transmission of raw data streams as byte arrays.
- **void**, used for indicating that no value is contained by the variable.
- **any**, means that any native type with which variable is initialized will be accepted.

Untyped subnodes, expressed as {?} construction, indicate that a node may have any kind of subtree. Already defined types can be reused in other types definition as link types by means of their `id`s.

Types may have any number of subtypes, which bnf-form is the following:

```
SubTypeList  ::=  SubType
             |    SubType SubTypeList

SubTypet     ::=  .id Cardinality : TypeDefinition
```

Each subnode has its cardinality defined as one by default or as following:

Cardinality	::=	[**int**,**int**]	(*Range*)
	|	[**int**,∗]	(*Lower − bound*)
	|	∗	(*Shortcut for* [0,∗])
	|	?	(*Shortcut for* [0,1])
	|	ϵ	

III. Case Study

In this section we show how the extension to the Jolie type system led to an enhanced arsenal at developers' disposal. We will proceed by examples. The short compendium of the Jolie syntax as presented in the previous section, combined with the code examples presented here, may be sufficient to grasp a general understanding. For a more comprehensive information the reader can refer to [16] and [6].

Let us consider an example implementing a car rental. This consists of three parts, as depicted in Figure III.1:

- *Server*, which provides the rental service;
- *Client*, which wants to use it;
- *Interface*, which declares the operations by means of which client and server can interact with each other.

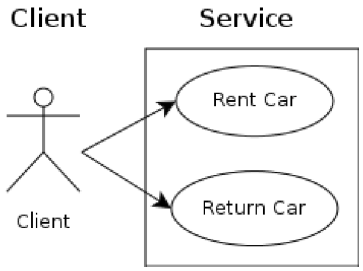

Figure III.1. Client-service use-case diagram

We will consider two possible approaches for the server to handle clients' requests: process-driven, implemented in Jolie language by means of input-guarded operations; and data-driven implemented by the newly added choice operator. We will then add some considerations on the usage of both.

A. Process-driven approach

In this example, the *interface* contains the definition of two operations dedicated to renting and returning the car, and several data types, as showed below:

- `customer`: stores customers personal information (name, age, and driving license number) necessary to rent a car;
- `car_return`: holds the reference to customer's profile who has rent this car, the car's identification number and the state of the car after renting.

```
//Car rent interface
type customer: void {
        .name: string
        .age: int
        .license: string
}

type car_return: void {
        .car_state: string
        .c?: customer
        .car_id: string
}

interface CarRentInterface {
          RequestResponse:
        get_car( customer )( string )
          RequestResponse:
        return_car( car_return )( string )
}
```

the server indicates the usage of operations defined in the above interface. This is done by the **include** directive at the beginning of the source code. The server's deployment part contains declaration of input port, by means of which it can

be accessed with the name and the protocol provided on the defined location. The behavior part of the program contains definition of two operations, `get_car` and `return_car`. They are placed inside the square brackets "[..]", which are used in input-guarded choice syntax. This means, that only one of the operations can be executed at the time, while the others will be deactivated.

```
//Server with input-guarded operations
include "carRentInterface.iol"

inputPort RentService {
      Location: "socket://localhost:2001"
      Protocol: sodep
      Interfaces: CarRentInterface
}

execution{ concurrent }

main{
      [get_car( request )( response ){
         response = "43535"
      }]

      [return_car( request )( response ){
         if (request.car_state == "damaged"){
            response = "Car is damaged!"
         } else {
            response = "Thank you!"
         }
      }]
}
```

The deployment part of the client describes how the connection to the Rent Service works: the same interface, protocol and Rent Service location. The behavioral part of the client program executes the following operations:

- creating the request information
- sending this information to Rent Service to be processed by `get_car` procedure
- checking the response and printing it out.

```
//Client.ol
include "carRentInterface.iol"

include "console.iol"

outputPort RentService {
      Location: "socket://localhost:2001"
      Protocol: sodep
      Interface: CarRentInterface
}

main{
      //sending request for a car
      customer.name = "John Smith";
      customer.age = 32;
      customer.license = "l23454675";

      get_car@RentService(customer)(response);
      println@Console
        ( "Car rent request is accepted. ")();
      println@Console
        ( "Car id is " + response )()

      //returnig the car
      return.car_id = response;
      return.car_state = "damaged";
      return_car@RentService( return )( response );
      println@Console
        ( "Car is returned. " + response )()
}
```

In order to better understand the execution, we show here the results of running the application:
Process-driven approach
Car rent request is accepted. Car id is 43535
Car is returned. Car is damaged!

The process-driven approach shows how, classically, the action flows is directed via the use of input-guarded choice. This approach is heavily influenced by the heritage of process algebra, as described in [6]. Input-guarded choice directed flow is indeed the basic mechanism in, for example, the untyped π-calculus [11]. Consequently, for this approach to be supported and implementable the language does not require to be particularly rich in term of type system and language primitives. Jolie itself originally supported only this mechanism.

B. Data-driven approach

In this case, the choice operator allows us to use a data-driven approach to computations.

Let's enrich the interface syntax with the following data type and operation:

```
//Car rent interface
...
type request: customer | car_return

interface CarRentInterface {
      ...
      RequestResponse:
        process(request)(string)
}
```

`process` is an operation that takes variable of `request` type, which itself can be either of `customer` or `car_return` types.

Let's implement the new server, supporting `process` operation[2]:

```
include "carRentInterface.iol"
```

[2]The `match` directive is not in the stable version of the project yet, so this example will not be compiled with the current (1.4.1) version of the interpreter.

```
inputPort RentService2 {
      Location: "socket://localhost:2002"
      Protocol: sodep
      Interfaces: CarRentInterface
}

execution{ concurrent }

main{
      process_user_request(request)(response){
      request match {
        customer { response = "43535" };
        car_return {
         if (request.car_state == "damaged"){
           response = "Car is damaged!"
         } else {
           response = "Thank you!"
         }
        }
      }
}
```

In this case we do not need to separate execution of operations by means of processes. Here it is done by means of the type of the input request variable.

In order to test the proposed approach, let's change the client program's code:

```
//Client.ol
include "carRentInterface.iol"
include "console.iol"

outputPort RentService {
      ...
}
outputPort RentService2 {
      Location: "socket://localhost:2002"
      Protocol: sodep
      Interfaces: CarRentInterface
}

main{
      //sending request for a car
      println@Console( "Process-driven approach")();
      ...

      //returing the car
      ...

      //working with server based on data type
      println@Console( "Data-driven approach")();
      process@RentService2(customer)(response);
      println@Console
        ( "Car rent request is accepted. ")();
      println@Console
        ( "Car id is " + response )();
      process@RentService2(return)(response);
      println@Console
        ( "Car is returned. " + response )()
}
```

The result of the execution will be the following:

Data-driven approach
Car rent request is accepted. Car id is 43535
Car is returned. Car is damaged!

The results are clearly the same than the process-driven approach, showing how behaviorally the two implementations appear indistinguishable for an external observer. However, as we have seen, the internal computations actually differ, and performances can differ too as long as other quality attributes. It is beyond the scope of this paper to make any quantitative assessment. It is enough to notice how enriched language mechanisms offer alternative programming pattern – and therefore design patterns – to developers who adopt the service-oriented paradigm. The development of specific design guidelines is left as future work.

IV. Architecture of Jolie interpreter

In order to explain better the changes that were applied to the interpreter, we have to describe how its basic components are working together.

The Jolie interpreter is written in Java, and its architecture is organized into several components [12], the most significant of which regard parsing the source files, establish the communication between components and running them, as depicted in Figure IV.1.

The *Parser* component scans and parses the source code, transforms and organizes it as a tree of objects with desired semantics. As a result, the parser produces *OOIT* (Object Oriented Interpretation Tree), which implements the execution of the semantic rules relative to the input program. *Runtime environment* instantiates other components and execute the *OOIT*. Finally, the *Communication core* part is in charge of performing communication between different components abstracting from the communication methods and protocols.

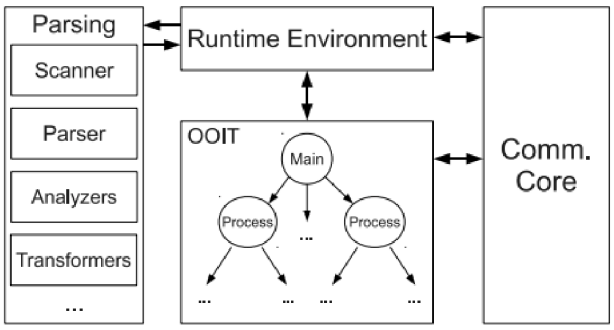

Figure IV.1. Jolie interpreter architecture

The following, discusses the interpreter's components in more details.

A. Parser

The parsing process follows the following stages and involves the following components:

1) *Scanner* reads the input and creates token objects based on the ones defined in token types enumeration;
2) *OLParser* is a recursive descent parser, which takes created tokens, checks them by the grammar rules and generates the corresponding syntax node in the abstract syntax tree (AST);
3) *OLParseTreeOptimizer* takes ready AST and optimizes it by reducing the number of nodes or transforming the code to more efficient versions;
4) *SemanticVerifier* checks whether the code is well-formed and semantically correct.

Both *OLParserTreeOptimizer* and *SemanticVerifier* use visitor design pattern [5] to access the AST (it implements the *OLVisitor* class).

Types mentioned in section II-D are expressed as nodes of AST. They are implemented by means of descendants of the abstract class *TypeDefinition*:

- *TypeInlineDefinition* for expressing native types;
- *TypeDefinitionLink* for linked types;
- *TypeDefinitionUndefined* for undefined types.

B. Runtime environment

OOITBuilder reads the AST and produces object tree-like data structure called object-oriented interpretation tree (*OOIT*). It defines the semantics for program execution. *OOIT* nodes implement Process interface, so that each node is responsible for the semantics of a single statement.

Runtime Environment handles parallel execution by means of native threads. Threads, responsible for executing a part of the *OOIT*, can be of two types:

1) Session threads, which are used for handling different sessions and retaining a local state for variable values.
2) Parallel threads, which are used for handling parallel composition and refering to their parent session thread for state handling.

C. Communication core

Communication Core component is used for performing communications by means of such mechanisms as messages and channels. Channels are used for the sending and receiving of messages, which consist of resource path, name of operation they are dedicated for, message content and in some cases fault name. Channels are in charge of encoding/decoding messages using the right protocol and sending/receiving them by means of the right communication medium.

V. Towards data-driven Workflows for Microservices

Jolie still experiences lack of some data types and operations, which could enrich its syntax and add extra flexibility, like, for example, regular expressions or choice operator. This work is dedicated to implementing the choice operator (as explained in Section V-A). It was implemented by extending the Jolie's type system. There were several architectural changes reported in Section V-B. Enriching Jolie's semantics with the choice operator adds extra flexibility to the language: there is no need to separate the execution of operations via processes, it can instead be discriminated by the type of the input request variable, as showed by the case study in Section III; it is possible to implement generic functions and behavior of functions based on arguments passed (see Section V-C).

A. The choice operator

The idea of choice operator was taken from XML [3]. Choice operator in XML allows to put several elements in choice element declaration, but to be presented with only one of them. For example, there is a choice element called "animal" in the example below, which can be either presented as a "dog" or a "cat":

```
<xs:element name="animal">
  <xs:complexType>
    <xs:choice>
      <xs:element name="cat" type="cat"/>
      <xs:element name="dog" type="dog"/>
    </xs:choice>
  </xs:complexType>
</xs:element>
```

Unlike the choice element in XML that enables users to choose between several types, in our implementation, this construction provides the possibility of choosing between two types only. However, this limitation can be easily removed. The same example will look in Jolie language in the following way:

```
type animal: cat | dog
```

We use pipe character ("|") for choice operator. Note, that in this case `cat` and `dog` are linked types and need to be declared explicitly, otherwise Jolie interpreter will raise an exception.

`TypeDefinition` formal grammar was enriched correspondingly:

```
TypeDefinition  ::=  ...
                |    TypeDefinition | TypeDefinition
```

B. Architectural changes

In this section we will describe the architectural changes necessary to implement the new data type.

1) Adding choice type to AST: In order to support the possibility of storing two types, *TypeChoiceDefinition* class was created (extends *TypeDefinition* as *TypeInlineDefinition* and others).

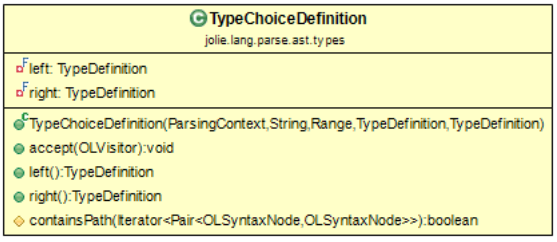

Figure V.1. TypeChoiceDefinition class diagram

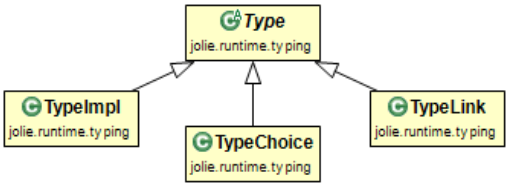

Figure V.2. Jolie type system class diagram

The *TypeChoiceDefinition* class (see Figure V.1) contains two attributes, *left* and *right*, representing the two possible types that can be any of *TypeDefinition* members. The addition of this class requires a change in the parent *TypeDefinition* class and its descendants, as they contain several methods that are not applicable to the choice type (like checking if a type has subtypes or its native type), which might raise null reference exception while trying to access them.

2) Extending parser: Extension of the parser requires working on *Scanner*, *OLParser*, *OLParserTreeOptimizer* and *SemanticVerifier*:

Scanner. No changes to the scanner are needed, because the pipe-symbol ("|"), which represents choice operator, is already in use as the parallel operator, so it is already present in the *TokenType* enum. Parallel operator, placed between operands, indicates that they are executed concurrently, as showed below

```
statementA | {statementB; statementC}
```

However, execution relates to the behavior part of the program, while message types have to be defined in the deployment part, and they are separately processed, so no effort on pipe symbol processing redefinition is needed.

OLParser parses message types and their subtypes by means of *parseTypes* and *parseSubTypes* methods correspondingly. These methods check that the sequence of tokens, generated by *Scanner*, is correct regarding the type definition grammar, as presented in Section II and then generate a node in the AST of one of the *TypeDefinition* descendants. Since this grammar has been enriched, *parseTypes* and *parseSubTypes* methods have been changed correspondingly, so that they could expect "|" token inside the type definition.

OLParserTreeOptimizer and *SemanticVerifier* classes work with the AST by means of visitors, so a visit method for *TypeChoiceDefinition* type was added to the *OLVisitor* class. In this case, no specific optimization is needed, so this visitor simply does the same actions as the other *TypeDefinition* class descendants.

SemanticVerifier checks whether the types with the same name have been already defined and checks cardinality of types. New visit method, added to deal with objects of *TypeChoiceDefinition* type, is doing the same, except that it also invokes semantic verification on both of its types inside.

3) Extending runtime facilities: Extending runtime facilities regards mostly building OOIT process and message types themselves.

Additions to building Object-Oriented Interpretation Tree process. OOITBuilder generates object oriented interpretation tree, based on the AST produced by the *OLParse*. It also uses visitors to access nodes of AST. Since new *TypeChoiceDefinition* type was added, a corresponding visitor was created, which is in charge of creating type object based on current *TypeChoiceDefinition* object. Type objects are discussed in the next section.

Message types. As messages are being passed by means of type objects, each of the *TypeDefinition* descendants in the AST should have a corresponding representation in the *Type* class descendants (*TypeImpl* for *TypeInlineDefinition*, *TypeLink* for *TypeDefinitionLink*). For representing *TypeChoiceDefinition*, a new *Type* descendant, *TypeChoice*, has been created, as depicted in Figure V.2.

C. Added flexibility to Jolie

1) Working with types and subtypes: Choice operator can work with native types:

```
type numeric: int | long
```

Linked types:

```
type linked_type: string
type linked_choice: linked_type | void
```

Subtypes:

```
person_info : void { .id: string | int}
```

2) Functions genericity and behavior based on arguments: One of the advantage of the choice operator is the possibility to create generic functions. If we declare the following choice type and function in interface

```
type choice : string | int
fun_choice ( choice )( choice )
```

we can provide the same behavior without considering which type (**string** or **int**) was passed to the function:

```
fun_choice(request)(response){
  response = response
}
```

Or imagine that we need to implement different behavior of the function based on the arguments passed. Let's have an interface with type `person`, able to be presented as element with type `personSSN` or `personCCN`:

```
type person: personSSN | personCCN
type personSSN : void {
  .ssn : int
}
type personCCN : void {
  .ccn : string
}
```

And a function `pay`, which takes as an input an argument of type `person` and runs the corresponding code based on a particular type (`personSSN` or `personCCN`) of the argument passed.

```
pay(person)(response) {
  person match {
    personCCN: ...
    |
    personSSN: ...
  }
  if ( is_defined( person.ssn ) ) {
```

```
        // ask the person registry
      } else {
        // contact the bank
      }
}
```

It is also possible to use the choice operator if users need to support several versions of data structures. For example, it was needed to process data related to `Old-Software` corporation, later its title has been changed to `New-Software`, but some customers can still use the old one.

```
type Old-Software-Corp: void {.
      .name: string
      .address: string
}
type New-Software-Corp: void {.
      .name: void {
      .firstname: string
      .lastname: string
      .address: string
      .phone: int
}
type corporation: Old-Software-Corp
      | New-Software-Corp
```

VI. Conclusions and Future work

Jolie is a comprehensive programming language based on the service-oriented paradigm [13], which emerged in the context of an extended research effort aimed at formalizing Service-Oriented Computing on top of broadly accepted models of concurrency in the EU Project SENSORIA (see, e.g., [6], [9]). Due to its support for the quick prototyping of both simple services and complex service coordination, Jolie has been used in the development of other research projects involving the programming and deployment of services (including [4], [15]). However, the language still lacks of data types and operations that would enrich its syntax and add extra flexibility to code common SOA scenarios: regular expressions and choice operator are just examples of this deficiency. The work presented in this paper has been devoted to extend the Jolie type system in order to add the choice operator and realize the necessary changes into the interpreter.

The major outcomes can be summarized as follows:

- Identification of data types able to support common SOA programming scenarios
- Addition of choice operator in the syntax and semantics of the language
- Analysis and reengineering of Jolie interpreter

This work shows how computation can move from a process-driven to a data-driven approach. Control-flow can be now directed at the time of message passing and according to the nature of the message structure and type, instead of requiring post-reception processing, hence leading to a preliminary identification of recurring communication patterns that can be then shaped as design patterns. In the microservice scenario this represents a novel opportunity opening to new programming scenarios.

Future work leaves space to both theoretical investigation and practical realization. Implementing regular expressions is a natural step in order to further enrich the type system and manage a broader set of programming scenarios. For what concerns theoretical aspects, formalization of the extended type system is considered a priority.

Acknowledgements

We would like to thank Innopolis University for logistic and financial support. This work was also partially supported by CRC (Choreographies for Reliable and efficient Communication software), grant no. DFF–4005-00304 from the Danish Council for Independent Research. Our gratitude goes to colleagues at IU who participated in discussions and seminars: Bertrand Meyer, Daniel de Carvalho, Mohamed Elwakil, Leonard Johard, Alexander Naumchev, Alexander Chichigin and Rasul Tumyrkin.

References

[1] Service Architecture: Service-Oriented Architecture (SOA) Definition. Accessed December 2015. http://www.service-architecture.com/articles/web-services/service-oriented_architecture_soa_definition.html, (2000-2015).

[2] Web Services Description Language (WSDL) Version 2.0 Part 1: Core Language. Accessed December 2015. http://www.w3.org/TR/wsdl20/, (2007).

[3] XML Schema choice Element. Accessed December 2015. http://www.w3schools.com/schema/el_choice.asp, (1999-2015).

[4] Maurizio Gabbrielli, Saverio Giallorenzo, and Fabrizio Montesi. Applied choreographies. *CoRR*, abs/1510.03637, 2015.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[6] Claudio Guidi, Roberto Lucchi, and Manuel Mazzara. A formal framework for web services coordination. *Electronic Notes in Theoretical Computer Science*, 180(2):55–70, June 2007.

[7] Claudio Guidi, Roberto Lucchi, Gianluigi Zavattaro, Nadia Busi, and Roberto Gorrieri. Sock: a calculus for service oriented computing. In *In ICSOC, volume 4294 of LNCS*, pages 327–338. Springer, 2006.

[8] J. Lewis and M. Fowler. Microservices. Accessed December 2015. http://martinfowler.com/articles/microservices.html, (2014).

[9] Roberto Lucchi and Manuel Mazzara. A pi-calculus based semantics for WS-BPEL. *J. Log. Algebr. Program.*, 70(1):96–118, 2007.

[10] Manuel Mazzara, Faisal Abouzaid, Nicola Dragoni, and Anirban Bhattacharyya. Toward design, modelling and analysis of dynamic workflow reconfigurations - A process algebra perspective. In Marco Carbone and Jean-Marc Petit, editors, *Web Services and Formal Methods - 8th International Workshop, WS-FM, Clermont-Ferrand, France, September 1-2, 2011*, volume 7176 of *LNCS*, pages 64–78. Springer, 2011.

[11] Robin Milner. *Communicating and Mobile Systems: The &Pgr;-calculus*. Cambridge University Press, New York, NY, USA, 1999.

[12] Fabrizio Montesi. JOLIE: a Service-oriented Programming Language. Master's thesis, University of Bologna, 2010.

[13] Fabrizio Montesi, Claudio Guidi, and Gianluigi Zavattaro. Service-oriented programming with jolie. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 81–107. Springer, 2014.

[14] J. M. Nielsen. A type system for the jolie language. Master's thesis, Technical University of Denmark, Department of Applied Mathematics and Computer Science / DTU Co, Matematiktorvet, Building 303B, DK-2800 Kgs. Lyngby, Denmark, compute@compute.dtu.dk, 2013. DTU supervisor: Nicola Dragoni, ndra@dtu.dk, DTU Compute.

[15] Mila Dalla Preda, Saverio Giallorenzo, Ivan Lanese, Jacopo Mauro, and Maurizio Gabbrielli. AIOCJ: A choreographic framework for safe adaptive distributed applications. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 161–170. Springer, 2014.

[16] The Jolie Team. The first language for Microservices. Accessed December 2015. http://www.jolie-lang.org/, (2014-2015).