An Abstract Machine for Integrating Heterogeneous Web Applications

Marcio A. Macedo*, Daniel A. S. Carvalho†, Martin A. Musicante*, Alberto Pardo‡ and Umberto S. Costa*
*Computer Science Department, Federal University of Rio Grande do Norte, Natal, Brazil
E-mail: marcioalves@ppgsc.ufrn.br, {mam, umberto}@dimap.ufrn.br
†Université Jean Moulin Lyon 3, Lyon, France, E-mail: danielboni@gmail.com
‡Instituto de Computación, Universidad de la República, Montevideo, Uruguay, E-mail: pardo@fing.edu.uy

***Abstract*—The adoption of Cloud Computing technologies by the organizations has profound consequences on the way software applications are developed and used. The migration to the Cloud may be accompanied by the revision of the business process, to integrate tasks over big data. In this way, existing workflow implementations may be extended with calls to operations in Hadoop or other tools. In this work, we define a framework to implement business process over heterogeneous technologies. Our framework is based on a novel workflow engine, called μBP-AM. Workflow execution in μBP-AM is performed by successively transforming the graph obtained from a workflow definition. μBP-AM has a formal semantics, which gives a precise definition of *how* the workflow is implemented. μBP-AM is at the core of an extensible framework capable of supporting not only Web service operations but also Hadoop operation calls (among others). The tool described here was conceived to increase reliability and to promote interoperability. We describe a prototype implementation of our framework, as well as some experimental results. Experiments using this prototype show that compositions run in μBP-AM using about the same resources as those run by using other tools.**

***Index Terms*—Interoperability, Abstract Machine, Big Data, Integration, Operational Semantics.**

I. Introduction

The need for better-defined behavioral interfaces for software components and Web services has motivated the definition of service composition languages. BPEL [1], ORC [2] and BPMN [3] are examples of languages to define composite Web services. Typically, the runtime support for this kind of languages provides the means to define and invoke Web services. The runtime environment itself is usually described in an informal way. This kind of languages are used to implement the business processes of private and public organizations around the world.

The advent of Cloud Computing [4], [5] and Big Data Processing [6] presents new challenges to the area of service compositions: Big Data operations are being integrated to the workflows implemented by the organizations, which increasingly relies on analytics. Service composition languages may be used in this new context by integrating them with Big Data processing tools like Hadoop [7]. One initiative in this area is the Pony system [8], which integrates Map/Reduce operations into a BPEL engine.

This work was partly funded by CNPq (Brazil, PDE-201118/2014-9, PQ-305619/2012-8, INCT-INES-573964/2008-4), CAPES/CNRS (Brazil/France, SticAmSud 052/14) and CAPES/ANII (Brazil/Uruguay, Capes-UdelaR 021/10).

As the use of computer (software) systems become more generalized, the concerns about their reliability also gain in importance. This is particularly true when the use of these applications may affect human lives or economic assets. Business process implementation may have a big impact on the economic resources of organizations and individuals, so, in many cases, they should be considered as critical. The use of formal specification techniques is one step towards improving reliability. The precise, unambiguous definition of a system may lead to a more reliable implementation, thus helping to bridge the gap between the specification and the implementation of a software system. In the case of a workflow system, the formal definition of its operations and constructors should provide a more detailed, precise guide for the implementation of the system, reducing possible interpretation errors. In this work, we propose the formal specification of a workflow implementation system. The advantages of our approach include the better understanding of the system, as well as the possibility of proving properties of workflows and of the implementation itself (in the case of critical systems), thus contributing to the reliability of the applications.

The main contributions of this paper are:

- A runtime system for the execution of workflows (expressed in a simple BPEL-like language). This runtime system is implemented as a *graph reduction machine*, where workflows are translated into graphs. These graphs are then transformed in accordance to formally defined reduction rules. We argue that the formal definition of this runtime system is useful both to provide a clear understanding of the workflow language, as well as to guide the implementation of the execution environment.
- The definition of an extensible, configurable tool for the implementation of workflows that are built upon different kinds of operations. In this way, a variety of operations may be seamlessly called from inside the workflow, depending only on the existence of a configuration file. So far, we have built interface support for Web Services, local (Python) procedures and Hadoop (including HDFS commands, MapReduce, PIG and Hive operations).
- A proof of concept implementation for our framework. We provide examples of use of the framework, as well as a comparison with other tools. Experiments with this prototype

978-1-5090-0478-2/15/$31.00 ©2015 IEEE

indicate that the use of resources for our tool is comparable with other implementations that are not formally specified.

This paper is organized as follows. Section II presents some related work. Section III and Section IV present a simple workflow language and the μBP-AM abstract machine. Our Proof-of-Concept implementation, as well as, some experimental results are given in Section V. Section VI is devoted to the conclusions of this work.

II. Related Work

Many approaches have been proposed for the development of Web service applications, ranging from contributions on theoretical foundations to the definition of middlewares. In this scenario, we can distinguish workflow languages conceived for business problems from those targeted at scientific problems [9]. In the business domain, routines between enterprises, business rules, process integrity and human involvement are major concerns. BPEL is one language for business applications. In the scientific domain, the presentation of results, efficiency on data processing and computation scheduling are central questions. Kepler [10] and Taverna [11] are workflow languages in this latter domain. Next, we present some work on the business domain, the perspective adopted in our proposal.

The formal treatment of Web services include the study of their properties using Process Algebra [12], [13], Automata [14], [15], Abstract State Machines [16], [17], Petri Nets [18], [19] among others. Although the formal treatment of Web service compositions is recurrent in the literature, the formalization of runtime systems for them is rare. A formally defined runtime system for ORC-like languages is proposed by Bruni, Nicola, Loreti and Mezzina [20]. That work defines an abstract machine that implements the operational semantics of the language. A similar work is presented by AlTurki and Meseguer [21], where an abstract machine for the execution of ORC-like languages is defined and implemented in Maude [22].

The Pony system [8] uses the BPEL language to compose Hadoop operations (instead of Web service operations). One goal of that work is to integrate both Hadoop and a BPEL engine without modifications. The authors propose the use of wrappers for each Hadoop operation, so that the BPEL engine sees them as Web service operations. The authors show that this technique is not only feasible but remains efficient, when compared with standard workflow engines for Hadoop, such as Oozie [23]. Although the Pony system and our work are intended to have similar use, Pony is not designed towards improving reliability. Another important difference is that Pony uses BPEL as its workflow language. Our proposal is intended to be more general, in the sense that any workflow language may be translated into the graphs supported by our engine.

III. A Simple Workflow Language

In this section we present a simple language for implementing workflows. This language is used as a general description tool and contains most of the elements present in popular workflow languages such as BPEL and BPMN. We refer to this language as μBPL. The language is represented by the following grammar:

$$
\begin{array}{rcl}
E & ::= & id \mid n \mid t \mid s \mid E_1 + E_2 \mid \ldots \mid E_1 \textbf{ and } E_2 \\
 & \mid & \ldots \mid E_1 == E_2 \mid \ldots \\
P & ::= & M(\overline{E};\overline{X}) \mid P_1 \parallel P_2 \mid P_1;P_2 \mid [E_1]P_1 + [E_2]P_2 \\
 & \mid & \textbf{if } E \textbf{ then } P \mid \textbf{while } E \textbf{ do } P
\end{array}
$$

The grammar rules for E describe the syntax of arithmetic and Boolean expressions formed from variable identifiers (id), numeric literals (n), Boolean literals (t), Strings (s) and the usual arithmetic, logic and relational operators.

The rules for P define the syntax of operation calls and workflow compositions:

- $M(\overline{E};\overline{X})$ represents the call to an operation M. $\overline{E}$ represents a sequence of expressions $E_1, \ldots, E_n$ that form the input arguments of M. $\overline{X}$ represents a sequence of identifiers $X_1, \ldots, X_k$ that will store the values returned by M. Notice that we adopt the convention that the symbol ";" separates input and output parameters. Input parameters precede output parameters in an operation call. For instance, a call to an operation $Add(a, b\ ;\ x)$ is interpreted as receiving the values of a and b as input and returning a value which will be stored in the variable x. Also, at this level, the operation name M represents all kinds of operations, including local ones, Web services, Hadoop, RMI or any other operation that will be executed using a black-box semantics. Notice that one or both of the lists $\overline{E}$ or $\overline{X}$ may be empty.
- $P_1 \parallel P_2$ and $P_1;P_2$ define, respectively, the interleaved and sequential composition of workflows P_1 and P_2.
- $[E_1]P_1 + [E_2]P_2$ is a guarded choice between P_1 and P_2. If both guards (E_1 and E_2) are true at the time of their evaluation, then a non-deterministic choice will be performed. Depending on the guards, this constructor may be used to represent both deterministic and non-deterministic alternatives.
- **if** E **then** P and **while** E **do** P represent, respectively, conditionals and loops.

The language defined above possesses enough generality to describe the workflows described by other languages such as BPEL and BPMN. Our language is very similar to that presented in [24], where we show a language that is capable of representing a large number of workflow patterns [25].

In the next section, this language will be used to generate workflow graphs, used by our abstract machine to execute the workflow.

IV. A Graph-Reduction Machine

The term *graph reduction* was coined by Wadsworth [26] to define a technique by which the graph representation of an expression is step-wisely transformed into another graph, representing the evaluation of the expression. In this context, each expression is represented by a rooted, directed graph. Graph transformation rules are represented by transition systems. The evaluation of a given expression is performed by iteratively applying these rules. Graph-reduction is usually implemented by an abstract machine. This sort of machines

were successfully used for the implementation of functional programming languages[1] and can be regarded as the predecessors of modern virtual machines such as the JVM [28].

The use of graphs to represent composite Web services is common to many middleware solutions.In most cases, a graph is used to describe the control or data flow of a composite service, and even to generate a program that implements the control or data dependencies. Despite this use of graphs for aiding at the definition of workflows, we are not aware of the use of a graph-reduction machine for the implementation of workflow composition infrastructures.

Let us now introduce μBP-AM, a graph-based abstract machine for the implementation of workflows. Our proposal consists of the translation of the workflow into a graph, which is successively transformed according to reduction rules. The transformation process represents the execution of the workflow. Specifically, our approach comprises two steps: *Translation* of a program into a graph and *Evaluation* of this graph by the successive application of transformation rules. Both steps are described next.

A. Translating Programs into Graphs

We define a function $\mathcal{T}$ that translates programs into their graph representations. In our setting, graphs are represented as pairs of sets $\langle V, A\rangle$. The set V is a set of nodes (or vertexes) and A is a set of directed edges (or arcs). Each vertex in V has a name as well as attributed data that is generated by the different cases of the translation function $\mathcal{T}$. We assume that function $\mathcal{T}$ generates unique vertex names. Given two nodes, named w and w', a directed arc from w to w' is noted $w \mapsto w'$.

The translation of composite programs is defined by induction on the structure of programs. Arcs of the graph are defined to reflect the execution order imposed by each workflow combinator. As usual, we use $indegree(v)$ (resp, $outdegree(v)$) which returns the number of incoming (resp, outgoing) arcs of a node v.

Operation invocation:: The graph corresponding to an operation call is defined as:

$$\mathcal{T}[\![M(\overline{E};\overline{X})]\!] = \langle\{w : M!\overline{E}, w' : M?\overline{X}\},\ \{w \mapsto w'\}\rangle.$$

It contains two nodes (w and w') and a single edge from w to w'. The information associated to w, written as $M!\overline{E}$, indicates that in order to process this node, a call to the operation M needs to be performed with input arguments $\overline{E}$. In the same way, the decoration of w' indicates that the result of the call to M needs to be awaited and that the values returned by M will be associated to the variables in $\overline{X}$. Notice that $\overline{E}$ is a list of expressions and $\overline{X}$ represents a list of variables. One or both of these lists may be empty. In all cases, the transition rules (Sect. IV-B) will implement *synchronous* operation calls.

Parallel composition:: The translation of a parallel composition $P_1 \| P_2$ is a graph whose set of nodes (resp. arcs) is the union of the sets of nodes (resp. arcs) of the graphs corresponding to P_1 and P_2.

$$\begin{array}{ll}\mathcal{T}[\![P_1 \| P_2]\!] = \langle V_1 \cup V_2,\ A_1 \cup A_2\rangle, & \\ \textbf{where} \quad \langle V_1, A_1\rangle = \mathcal{T}[\![P_1]\!], & \langle V_2, A_2\rangle = \mathcal{T}[\![P_2]\!].\end{array}$$

Sequential composition:: The translation of a sequential composition $P_1; P_2$ is a graph formed by the nodes of the graphs corresponding to P_1 and P_2.

$$\begin{array}{ll}\mathcal{T}[\![P_1; P_2]\!] = \langle V_1 \cup V_2,\ A_1 \cup A_2 \cup A'\rangle, & \\ \textbf{where} \quad \langle V_1, A_1\rangle = \mathcal{T}[\![P_1]\!], & \langle V_2, A_2\rangle = \mathcal{T}[\![P_2]\!], \\ & A' = \{w \mapsto w' | w \in V_1, outdegree(w) = 0, \\ & \qquad w' \in V_2, indegree(w') = 0\}\end{array}$$

The set of edges of the new graph is formed by the edges of the graphs for P_1 and P_2, together with new edges, from each anti-root of $\mathcal{T}[\![P_1]\!]$ to each root of $\mathcal{T}[\![P_2]\!]$.

Guarded choice:: A guarded choice $[E_1]P_1 + [E_2]P_2$ is translated into a graph formed by the union of the graphs corresponding to P_1 and P_2 plus three new nodes and some new arcs, as follows:

$$\begin{array}{ll}\mathcal{T}[\![[E_1]P_1 + [E_2]P_2]\!] = \langle V_1 \cup V_2 \cup V',\ A_1 \cup A_2 \cup A'\rangle, & \\ \textbf{where} \quad \langle V_1,\ A_1\rangle = \mathcal{T}[\![P_1]\!], & \langle V_2,\ A_2\rangle = \mathcal{T}[\![P_2]\!], \\ & V' = \{v : +, v_1 : E_1, v_2 : E_2\}, \\ & A' = \{v \mapsto v_1, v \mapsto v_2\} \cup \Delta_1 \cup \Delta_2, \\ & \Delta_1 = \{v_1 \mapsto w \mid w \in V_1, indegree(w) = 0\}, \\ & \Delta_2 = \{v_2 \mapsto w \mid w \in V_2, indegree(w) = 0\}\end{array}$$

The new nodes correspond to the "+" operator and the two expressions E_1 and E_2. The generated graph has edges linking the "+" node to the guard nodes, as well as arcs linking each of the guard nodes to the roots of their guarded graphs.

Conditional:: The translation of a conditional **if** E **then** P is similar to that of a guarded choice, where one alternative is guarded with the condition E and the other one is always *false*.

$$\begin{array}{ll}\mathcal{T}[\![\textbf{if } E \textbf{ then } P]\!] = \langle V \cup V',\ A \cup A'\rangle, & \\ \textbf{where} \quad \langle V,\ A\rangle = \mathcal{T}[\![P]\!], & \\ & V' = \{v : +, v_1 : E, v_2 : \mathit{false}\}, \\ & A' = \{v \mapsto v_1,\ v \mapsto v_2\} \cup \Delta, \\ & \Delta = \{v_1 \mapsto w \mid w \in V, indegree(w) = 0\}\end{array}$$

While loops:: The translation of a while-loop workflow defines a graph with just one node. This node represents the whole iteration and will be expanded as needed during the execution of the workflow. The attribute $E.P$ contained at the node v represents the condition and the workflow present at the body of the loop. The expansion of the while nodes is implemented using just-in-time compilation [29].

$$\mathcal{T}[\![\textbf{while } E \textbf{ do } P]\!] = \langle\{v : E.P,\ \emptyset\}\rangle$$

The graph produced by the translation function $\mathcal{T}$ will be reduced according to the rules defined next.

[1] See, for instance [27].

B. μBP-AM Graph Transformation Rules

Given a workflow P, the graph $G = \mathcal{T}[\![P]\!]$ is transformed by the successive application of reduction rules in the context of an abstract architecture where G is one of its components. A *configuration* of μBP-AM is defined as a 4-tuple $\langle G, \rho, I, O\rangle$, where G is a graph, $\rho : id \rightarrow Val$ is an environment (a finite mapping from variables to values), and I and O are input and output buffers. Both buffers are lists of triples (o, v, ℓ) formed by an operation name o, a vertex name v, and a list of values ℓ. In the case of the input buffer I, the list ℓ corresponds to the values returned by a call to the operation o. In the case of the output buffer O, it corresponds to the values to be sent to the operation o at its invocation. It is worth mentioning that we are assuming the existence of a component external to the abstract machine that communicates with the outside world. Such a component manages the input and output buffers, performing the required operation calls appearing in O and placing the respective operation responses in I. This component is defined in Sect. V as the *Operation Manager* of the framework in Fig. 2.

The execution of μBP-AM follows a transition relation between configurations:

$$\langle G, \rho, I, O\rangle \ \triangleright \ \langle G', \rho', I', O'\rangle$$

This relation is defined ahead in this section.

The *initial configuration* for a program P is given by the 4-tuple $\langle \mathcal{T}[\![P]\!], \emptyset, \epsilon, \epsilon\rangle$, representing a configuration with empty environment $\emptyset$ and empty buffers ϵ. A *computation sequence* for a program P is defined as a sequence of configurations starting from the initial configuration such that every two consecutive configurations in the sequence c_i, c_{i+1} are in the transition relation: $c_i \triangleright c_{i+1}$. A computation sequence may be *terminating* if it is finite, or *looping* when it is infinite. When a computation sequence is terminating it may end in a terminal configuration (a configuration with an empty graph), which is referred to as *normal termination*, or in a stuck configuration, also called an *abnormal termination*. There are many situations that may lead to an abnormal termination. For example, when the program is in deadlock, caused by mutual data dependencies between parallel branches (characterizing a circular wait) .

Let us now define the reduction rules corresponding to the transition relation. For any set R, such that $x \notin R$ we write $R[x]$ to denote $R \cup \{x\}$.

a) Operation call:: Recall that, in the graph, a call to an operation M is represented by a vertex w decorated with $M!\overline{E}$, being $\overline{E}$ a sequence of expressions, corresponding to the arguments of the call to M. We know, by the graph construction, that w must be followed by a node w' which is decorated with the response of M, and that both nodes are linked by an edge of the graph. In order to invoke the operation M, μBP-AM verifies that *(i)* w is a root node of the graph (that is, there is no incoming arc to that node) and *(ii)* all the variables on the expressions in $\overline{E}$ are already assigned (that is, they are associated to a value in the environment ρ, noted by $Vars(\overline{E}) \subseteq Dom(\rho)$).

In an operation call, μBP-AM writes the triple $(M, w', \overline{a})$ to the output buffer O. In this triple, $\overline{a}$ is the list of values to be passed as arguments in the call to M. Those values are obtained by the evaluation of the expressions in $\overline{E}$ using the variable environment ρ (noted $\overline{a} = Eval_\rho(\overline{E})$). The reduction rule for this case is the following:

$$\frac{\begin{array}{c} indegree(w) = 0, \quad Vars(\overline{E}) \subseteq Dom(\rho), \quad \overline{a} = Eval_\rho(\overline{E}), \\ G = \langle V[w : M!\overline{E},\ w' : M?\overline{X}], A[w \mapsto w']\rangle \end{array}}{\langle G, \rho, I, O\rangle \ \triangleright \ \langle\langle V[w' : M?\overline{X}], A\rangle, \rho, I, O[(M, w', \overline{a})]\rangle}$$

The identification w' is used as a unique identifier, to be associated to the response to the call. In this way, our architecture implements a key-based correlation [30], in the case of operation calls.

Notice that, one or two of the lists $\overline{E}$ or $\overline{X}$ may be empty. As our abstract machine defines synchronous communication, the execution of part of workflow will be blocked until the operation finishes its execution. This restriction will be relaxed in future versions of our abstract machine, making it possible to natively implement asynchronous operation calls.

Operation Response:: The answer to an operation call M causes, at some later time, the appearance of a response in the input buffer I of the abstract machine. Once a response $(M, w, \overline{r})$ arrives at the input buffer, the machine has to wait until w is a selected root node to bind the returned values $\overline{r}$ to the variables $\overline{X}$ specified in the operation response. When that is the case, the returned values are bound to the corresponding variables in the environment ρ, and the node w is removed from the graph, together with all its outbound edges:

$$\frac{\begin{array}{c} indegree(w) = 0, \quad \overline{X} = (x_1, \ldots, x_k), \quad \overline{r} = (v_1, \ldots, v_k), \\ \rho' = \rho \cup \{(x_i, v_i) | 1 \leq i \leq k\}, \ A' = A - \{w \mapsto v | v \in V\} \end{array}}{\langle\langle V[w : M?\overline{X}], A\rangle, \rho, I[(M, w, \overline{r})], O\rangle \ \triangleright \ \langle\langle V, A'\rangle, \rho', I, O\rangle}$$

Notice that, the list $\overline{X}$ may be empty. Also, the number of elements of $\overline{X}$ must be the same as the number of elements of $\overline{r}$.

Guarded Alternatives:: The following rule defines the reduction of a graph in which a root node v represents a choice and one of its guards evaluates to true. In this case, the other alternative needs to be erased from the graph. We consider w being the node of the guard evaluated to true, whereas w' is the root of the subgraph to be erased.

$$\frac{\begin{array}{c} indegree(v) = 0, \ Vars(E) \subseteq Dom(\rho), \ Eval_\rho(E) = \mathbf{true}, \\ V' = \{u \in V \mid w' \mapsto^* u \wedge w \not\mapsto^* u\}, \ G = \langle V, A\rangle \setminus V' \end{array}}{\begin{array}{r} \langle\langle V[v : +, w : E, w' : E'], A[v \mapsto w, v \mapsto w']\rangle, \rho, I, O\rangle \\ \triangleright \langle G, \rho, I, O\rangle \end{array}}$$

In the rule above, the new graph G is obtained by removing those nodes belonging to the discarded alternative (as well as their corresponding edges), characterized by V' (i.e, those nodes of the graph that are reachable from w' but are not reachable from w). Notice that the alternative may contain just a node **false**, in the case of a conditional. The nodes v and w are also eliminated, as well as their outbound edges.

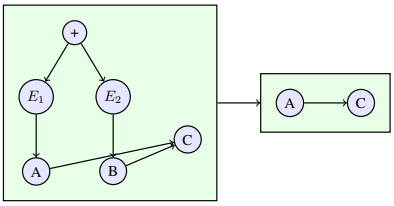

Fig. 1: Reduction of $([E_1]A + [E_2]B); C$ into $A; C$.

As an example[2], consider the program $([E_1]A + [E_2]B); C$ where a choice between two operations A and B is followed by a operation C. Let us suppose that during the execution of that program, the condition E_1 is true and A is the selected branch. Then, the graph that remains after reduction is the one composed by the graphs for A and C together with the arcs that connect them (see Fig. 1).

The next rule considers the case where a root node v represents a choice and all its guards are evaluate to **false**. In this case, we have to remove the sub-graphs corresponding to both alternatives, which are rooted at nodes w and w'. Notice that the nodes u to be eliminated from the graph are those that *(i)* are reachable from w and w' and *(ii)* are *not* reachable from both of them (this condition represents the case where the whole guarded command is part of a sequential composition).

$$\frac{\begin{array}{c} indegree(v) = 0, \qquad Vars(E) \cup Vars(E') \subseteq Dom(\rho), \\ Eval_\rho(E) = Eval_\rho(E') = \mathbf{false}, \\ V' = \{u \in V \mid w \mapsto^* u \wedge w' \not\mapsto^* u\}, \\ V'' = \{u \in V \mid w' \mapsto^* u \wedge w \not\mapsto^* u\}, \\ G = \langle V, A\rangle \setminus (V' \cup V'') \end{array}}{\begin{array}{r} \langle\langle V[v : +, w : E, w' : E'], A[v \mapsto w, v \mapsto w']\rangle, \rho, I, O\rangle \\ \rhd \langle G, \rho, I, O\rangle \end{array}}$$

While loops:: This rule is applied when a root node of the graph is a vertex $v : E.P$. This node will be simply expanded into a graph corresponding to a conditional expression involving the loop, unfolding one loop at a time. Indeed, the rule below implements the following transformation:

$$\mathbf{while}\ E\ \mathbf{do}\ P \rightarrow \mathbf{if}\ E\ \mathbf{then}\ (P; \mathbf{while}\ E\ \mathbf{do}\ P)$$

Notice that we use a (very simple) technique of just-in-time compilation [29].

$$\frac{\begin{array}{c} indegree(v) = 0, \qquad \langle V', A'\rangle = \mathcal{T}[\![P]\!], \\ V'' = V[v : E.P] \cup \{p : +, w : E, w' : false\} \cup V', \\ R = \{w \mapsto u | u \in V', indegree(u) = 0\}, \\ Q = \{u \mapsto v | u \in V' \wedge outdegree(u) = 0\}, \\ A'' = A \cup A' \cup \{p \mapsto w, p \mapsto w'\} \cup R \cup Q \end{array}}{\langle\langle V[v : E.P], A\rangle, \rho, I, O\rangle \rhd \langle\langle V'', A''\rangle, \rho, I, O\rangle}$$

In the rule above, the set R contains arcs from the node containing the expression E to all the roots of the graph of $\mathcal{T}[\![P]\!]$, thus, implementing the conditional. The set Q contains edges from the anti-roots of $\mathcal{T}[\![P]\!]$ to the node representing the while-loop. These edges implements the sequence inside the $\mathbf{if - then}$.

[2]We show this case, since it exemplifies the most intricate rule of μBP-AM. We do not show examples of other cases due to space restrictions.

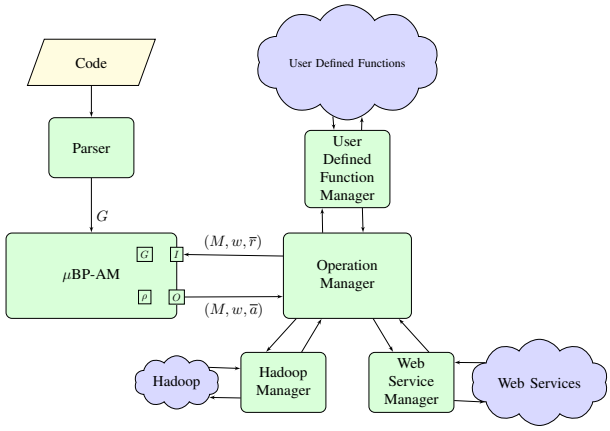

Fig. 2: μBP-AM architecture.

The following section describes the implementation of this abstract machine, as well as the design of the framework in which it is contained.

V. Implementation and Experimental Results

In this section, we briefly describe a framework for the implementation of workflows. The architecture of our framework is shown in Fig. 2.

The main components of our framework are a *parser* (to transform μBPL programs into graphs), a *reduction core* (implementing the abstract machine) and an *operation manager* (responsible for the interface to other systems, like Hadoop, RMI, etc). Our implementation was written in Python [31]. We used PLY (Python Lex-Yacc) [32] to generate the parser. The graph implementation uses NetworkX [33].

The operation manager is built to provide extensibility to our framework: It relies on the existence of a set of interface configuration files, defining the way each operation call (and response) should be processed. For example, given the configuration file in Fig. 3, the following call is a valid operation call and can be used inside a μBPL program: `ws.weather.getTemp("teresina","brazil"; T)`.

In this case, the configuration file defines that operation names beginning with ws correspond to Web service operations. The operation $getTemp$ of the Web service $weather$ is also specified in the configuration file. In addition to the ones defined in Fig. 3, interfaces to other technologies (such as SQL, RPC, etc) may be developed at a minimum effort.

Example 1. Let us suppose a weather monitoring system that has to post alerts on Twitter when temperature reaches certain conditions. The system monitors temperature in a given region and calculates the average temperature of three different cities. Temperature measurement is performed at five seconds intervals. The average temperature is published, independently, every ten seconds. The system issues alerts whenever the average temperature is outside the 10 to 25 degrees range.

```
SETTINGS = {
 "webservice": {
  "module": "mbpel.om.ws.webservice.WebServiceManager",
  "prefix": "ws",
  "services": {
   "weather": {
    "wsdl": "http://marcioalves.com.br:8080/Weather
                                /services/Weather?wsdl",
    "operations": ["getTemp"]
   },
 },
},
 "hadoop": {
  "module": "mbpel.om.hd.hadoop.HadoopManager",
   "prefix": "hd",
   "clusters": {
      "vm": {
        "host": "127.0.0.1",
        "port": "2222",
        "user": "root",
        "pass": "hadoop",
        "operations": ["pig", "ssh", "hdfs"],
      },
   },
 },
}
```

Fig. 3: Example configuration file.

The orchestration shown in Fig. 4 implements the monitoring system[3]. Our implementation is based on four services:

Storage: provides means to store, read and update variables. This service has three operations: `initVar(String name;)` that creates an integer variable (called `name`), initialized to 0; `load(String name; int value)` that returns the value stored in the variable `name`; and `store(String name, int value;)` that updates its value.

Delay: is a general service for time intervals. This service has only one operation `wait(int seconds;)`.

Weather: uses an external service called *GlobalWeather*[4] to return the temperature for a specific city. This service exposes the operation, `getTemp(String city, String country ; int value)` which returns the temperature for a given city.

Twitter: posts messages on *Twitter* using the operation `tweet(String text;)`.

In order to evaluate our implementation, we performed two different experiments: *(i)* Evaluation of the dynamic behavior of the prototype for the example program and *(ii)* Comparison with an implementation that uses an industrial middleware system and a implementation of a graph reduction machine in Java.

In the first experiment, we considered the dynamic evolution of some parameters of the generated graphs. We used the graph translated from the example in Fig. 4 to observe the evolution of:

[3]The program in Fig. 4 is presented as a didactic example, aimed to contain most of the constructors of the μBPL language. Notice that the line "`if ((AVG < 10) or (AVG > 25)) then`" may be omitted from the program.

[4]http://www.webservicex.com/globalweather.asmx?wsdl

```
ws.storage.initVar("average";);
while true do
  ( ws.weather.getTemp("teresina","brazil"; T)
  || ws.weather.getTemp("natal","brazil"; N)
  || ws.weather.getTemp("fortaleza","brazil"; F)
  );
  ws.storage.store("average", (T+N+F)/3; );
  ws.delay.waitTime(5;)
||
while true do
  ws.storage.load("average"; AVG);
  (if (AVG < 10) or (AVG > 25) then
    [AVG < 10] ws.twitter.tweet(AVG+": temperature too low";)
    +[AVG > 25] ws.twitter.tweet(AVG+": temperature too hight";)
  );
  ws.delay.waitTime(10;)
```

Fig. 4: Weather monitoring system.

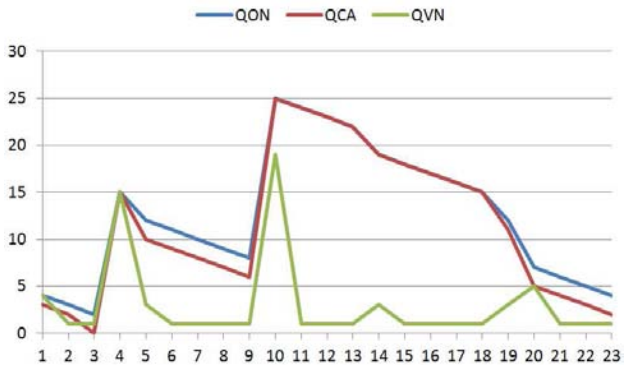

Fig. 5: Measurements for μBP-AM.

QON: Is the *Quantity of Nodes* of the graph that are reachable from the roots;

QCA: Is the *Quantity of Control Arcs* between nodes of the graph being reduced;

QVN: Is the *Quantity of Visited Nodes*, that is, the number of nodes that are handled by the machine in a reduction step.

These parameters were measured for the execution steps of the program in Fig. 4. Notice that the example consists on two loops that run forever in parallel. Owing to the non-terminating nature of the example, we considered those steps of the execution from the initial graph until one iteration of each loop is completed. The columns in Fig. 5 correspond to the states of the machine after a reduction step.

As the reduction rules are applied, the number of nodes and edges tends to decrease. The application of the *while* reduction rule is the only reduction that creates new nodes on the graph. In our example, the *while* operation is performed at steps 4 and 10. These steps correspond to the application of the reduction rule for *while*, where the *while* node is replaced by a fresh new instance of the entire graph of the loop. This also produces an increase on the number of visited nodes in those reduction steps, as we count created new nodes as visited.

On the first step, all the nodes of the graph were visited

TABLE I: Results for μBP-AM, PEWS-AM and PEWS-RT

Parameters	μBP-AM	PEWS-AM	PEWS-RT
LOC	345	617	670
QOM	29	42	126
NOA	15	102	69
CBO	13	30	119
QFC	332	1730	49

in order to identify its roots[5]. The steps 3 to 23 show the evolution of one iteration inside the parallel constructor. The values at both steps indicate that the machine is in similar states.

The second experiment has been devised to compare our implementation with two other: PEWS-AM and PEWS-RT. PEWS-AM [34] a graph reduction machine for Web service compositions built in Java. PEWS-AM is a predecessor of our abstract machine. PEWS-RT [35], an implementation of the μBPL language, for Web service compositions, built on Windows Workflow Foundation [36].

We performed the measurement of the following parameters, for the same example program of Fig. 4:

LOC: *Lines of Code* of the program generated by the translation from μBPL. The μBP-AM translator generates Python code, while PEWS-AM generate Java code and PEWS-RT generates C# code. Only the generated code was considered (excluding libraries and comments);
QOM: *Quantity of Methods* created by the translation;
NOA: *Number of Attributes* appearing in the classes generated by the translation;
CBO: *Coupling Between Objects* is the number of classes that are coupled to other classes. It is the number of classes referenced inside the generated programs;
QFC: *Quantity of Function Calls* inside the generated programs.

The Table I shows the results obtained from the measurement of the parameters LOC, QOM, NOA, QFC and CBO for Python, C# and Java documents generated by the tools μBP-AM, PEWS-RT and PEWS-AM respectively, for the program in Fig. 4.

We can see that our approach have a less number of lines of code (LOC) and attributes (NOA). This indicates that our implementation generates programs that are structurally comparable with code generated with industrial tools, but with a reduced number of lines of code and attributes.

Our approach has less methods (QOM) and coupling between objects (CBO) than PEWS-AM and PEWS-RT. This is an indication that our approach is conceptually simpler than the others, generating less complex interactions between objects. On the other hand, μBP-AM generates more function calls (QFC) than PEWS-RT, but less than PEWS-AM. These number of function calls were expected due to the continuous application of reduction rules by μBP-AM.

[5]This step can be improved if the list of roots is generated at the translation phase. We have chosen the prototype to be faithful to the translation functions presented in Sect. IV.

```
while true do
  ( ws.weather.getTemp("teresina","brazil"; T)
  || ws.weather.getTemp("natal","brazil"; N)
  || ws.weather.getTemp("fortaleza","brazil"; F)
  );
  my.operations.average(T,N,F; AVG);
  (if (AVG < 10) or (AVG > 25) then
    [AVG < 10] my.operations.appendToHadoopFile(AVG,
                    "/hadoopstorage/in/low_average.csv";)
    +[AVG > 25] my.operations.appendToHadoopFile(AVG,
                    "/hadoopstorage/in/hight_average.csv";)
  );
  my.operations.getTime(;HOUR, MINUTE, SECOND);
  if (HOUR==0 and MINUTE ==0 and SECOND==0) then
    hd.vm.mapreduce("temperature.jar", "Analysis",
            "/hadoopstorage/in/", "/hadoopstorage/out/";)
  ws.delay.waitTime(10;)
```

Fig. 6: Modified weather monitoring system.

Example 2. We now modify the weather monitoring system shown in the previous example so that, in addition to Web service operations, we use user-defined and Hadoop operations. Like before, the system monitors the temperature in a given region and calculates the average temperature of three different cities. The average temperature is now calculated using a user-defined function called "`average`". Another new feature of the system is that it sends data to Hadoop whenever the average temperature is outside the 10 to 25 degrees range. At midnight, using Hadoop's "`mapreduce`" operation, we analyze the data and finally wait for 10 seconds before measuring the temperatures of the cities again. The orchestration shown in Fig. 6 implements the modified monitoring system. This implementation uses the following modules:

operations: provides user-defined operations. This module has three operations:

```
average(int x, int y, int z; int avg)
```

which calculates the average of temperatures `x`, `y` and `z` and returns the result in the variable `avg`;

```
appendToHadoopFile(int avg, String path;)
```

which sends the `avg` data to a Hadoop cluster; and

```
getTime(;int hour, int minute, int second)
```

which return the current time in three variables: `hour`, `minute` and `second`.
vm: has a set of operations used by Hadoop. The only operation we use in this example is `mapreduce`, which performs the MapReduce operations of the jar file "`temperatures.jar`" saving the results into a directory on the Hadoop file system.

The prefixes `ws`, `my` and `hd` make reference to the prefixes informed in the configuration file. These prefixes indicates to the operation manager how to treat each operation call. The `ws` prefix states that the operation is a call to a Web service, `my` to user-defined functions and `hd` to Hadoop operations.

VI. Conclusions

In this paper we developed the idea of using a graph reduction machine for the execution of compositions of general

operations, including Web service operations, local programs and Hadoop operations. The orchestration language we use includes common constructors for Web service compositions and it is capable of expressing most control workflow patterns [24]. Our abstract machine μBP-AM compiles programs in this language into graph representations. The execution of programs is performed by the application of a set of formally specified graph reduction rules.

The abstract machine proposed here was implemented in Python. As a proof of concept, we provided interfaces to execute Web service operations, Python subprograms and Hadoop operations. Interfaces with other systems may be defined without difficulty.

In order to evaluate our implementation we measured a set of static and dynamic parameters. The comparison with other work shows that we can achieve a high degree of extensibility without sacrificing performance. We also observe that there is room for improvement of our proposal. In particular, we plan to reduce the number of function calls by substituting the busy-waiting on operation calls by a locking mechanism.

On the formal side, we plan to work on the definition of an operational semantics for our orchestration language and the proof that the abstract machine is correct with respect to that semantics.

Acknowledgments: We would like to thank Oliver Kopp for his insightful comments and suggestions on an early version of this paper.

REFERENCES

[1] OASIS, "Web Services Business Process Execution Language Version 2.0," OASIS Web Services Business Process Execution Language (WSBPEL) TC, Tech. Rep., Apr. 2007. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html

[2] D. Kitchin, A. Quark, W. R. Cook, and J. Misra, "The Orc Programming Language," in *FMOODS/FORTE 2009*, ser. LNCS, vol. 5522. Springer, 2009, pp. 1–25.

[3] O. M. G. (OMG), "Business Process Model and Notation (BPMN) Version 2.0," Object Management Group (OMG), Tech. Rep., jan 2011.

[4] F. Leymann, "Cloud Computing: The Next Revolution in IT," in *Proc. 52th Photogrammetric Week*. Wichmann Verlag, September 2009, Conference Paper, pp. 3–12.

[5] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A View of Cloud Computing," *Commun. ACM*, vol. 53, no. 4, pp. 50–58, Apr. 2010.

[6] J. Dean and S. Ghemawat, "MapReduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.

[7] A. S. Foundation, "Hadoop powered by," Feb. 2014, http://wiki.apache.org/hadoop/PoweredBy.

[8] J. Liu, Q. Li, F. Zhu, J. Wei, and D. Ye, "Building an Efficient Hadoop Workflow Engine Using BPEL," in *Current Trends in Web Engineering*. Springer, 2013, pp. 281–292.

[9] W. Tan, P. Missier, I. Foster, R. Madduri, D. De Roure, and C. Goble, "A comparison of using Taverna and BPEL in building scientific workflows: the case of caGrid," *Concurrency and Computation: Practice and Experience*, vol. 22, no. 9, pp. 1098–1117, 2010.

[10] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *Scientific and Statistical Database Management, 2004. 16th International Conference on*. IEEE, 2004, pp. 423–424.

[11] K. Wolstencroft, R. Haines, D. Fellows, A. Williams, D. Withers, S. Owen, S. Soiland-Reyes, I. Dunlop, A. Nenadic, P. Fisher *et al.*, "The Taverna workflow suite: designing and executing workflows of Web Services on the desktop, web or in the cloud," *Nucleic Acids Research*, vol. 41, no. Webserver-Issue, pp. 557–561, 2013.

[12] M. Weidlich, G. Decker, and M. Weske, "Efficient Analysis of BPEL 2.0 Processes Using p-Calculus," in *APSCC*. IEEE, 2007, pp. 266–274.

[13] K. Xu, Y. Liu, and G. Pu, "Formalization, verification and restructuring of bpel models with pi calculus and model checking," *Computer Science, IBM Research Report*, 2006.

[14] A. Wombacher, P. Fankhauser, and E. J. Neuhold, "Transforming BPEL into Annotated Deterministic Finite State Automata for Service Discovery," in *ICWS*. IEEE Computer Society, 2004, pp. 316–323.

[15] R. Kazhamiakin and M. Pistore, "A parametric communication model for the verification of bpel4ws compositions," in *In Mario Bravetti, Lela Kloul, and Gianluigi Zavattaro, editors, EPEW/WS-FM, volume 3670 of Lecture Notes in Computer Science*. Springer, 2005, pp. 318–332.

[16] R. Farahbod, U. Glässer, and M. Vajihollahi, "A Formal Semantics for the Business Process Execution Language for Web Services," in *WSMDEIS*, S. Bevinakoppa, L. F. Pires, and S. Hammoudi, Eds. INSTICC Press, 2005, pp. 122–133.

[17] D. Fahland, *Complete abstract operational semantics for the web service business process execution language*, ser. Informatik-Berichte / Institut fr Informatik, Humboldt Universitt zu Berlin. Berlin: Inst. fr Informatik, 2005, no. 190.

[18] K. Schmidt and C. Stahl, "A Petri net semantic for BPEL4WS - validation and application," in *Proceedings of the 11th Workshop on Algorithms and Tools for Petri Nets (AWPN'04)*, E. Kindler, Ed. Universitt Paderborn, 2004, pp. 1–6.

[19] H. Schlingloff, A. Martens, and K. Schmidt, "Modeling and Model Checking Web Services," *ENTCS*, vol. 126, pp. 3–26, 2005.

[20] R. Bruni, R. Nicola, M. Loreti, and L. Mezzina, "Provably Correct Implementations of Services," in *Trustworthy Global Computing*, ser. Lecture Notes in Computer Science, C. Kaklamanis and F. Nielson, Eds. Springer Berlin Heidelberg, 2009, vol. 5474, pp. 69–86.

[21] M. AlTurki and J. Meseguer, "Dist-Orc: A Rewriting-based Distributed Implementation of Orc with Formal Analysis," in *RTRTS*, ser. EPTCS, P. C. Ölveczky, Ed., vol. 36, 2010, pp. 26–45.

[22] N. Martí-Oliet, "An Introduction to Maude and Some of Its Applications," in *PADL*, ser. Lecture Notes in Computer Science, M. Carro and R. Peña, Eds., vol. 5937. Springer, 2010, pp. 4–9.

[23] M. K. Islam and A. Srinivasan, *Apache Oozie*. O'Reilly, 2014, early Release Ebook.

[24] M. G. Hahn, R. Motz, A. Pardo, and M. A. Musicante, "Formal Semantics and Expressiveness of a Web Service Composition Language," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, ser. SAC '13. New York, NY, USA: ACM, 2013, pp. 1667–1673.

[25] W. M. P. Van Der Aalst, A. H. M. Ter Hofstede, B. Kiepuszewski, and A. P. Barros, "Workflow patterns," *Distrib. Parallel Databases*, vol. 14, no. 1, pp. 5–51, Jul. 2003.

[26] C. Wadsworth, "Semantics and Pragmatics of Lambda-calculus," Ph.D. dissertation, Oxford University, 1971.

[27] T. Johnsson, "Efficient compilation of lazy evaluation," *SIGPLAN Notices*, vol. 39, no. 4, pp. 125–138, 2004.

[28] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley, "The Java Virtual Machine Specification," 2013, http://docs.oracle.com/javase/specs/jvms/se7/html/.

[29] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006.

[30] A. P. Barros, G. Decker, M. Dumas, and F. Weber, "Correlation Patterns in Service-Oriented Architectures," in *FASE*, ser. LNCS, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer, 2007, pp. 245–259.

[31] P. S. Foundation, "Python," Jan. 2015, https://www.python.org/.

[32] D. M. Beazley, "Ply, python lex-yacc (2001)," Jan. 2015, http://www.dabeaz. com/ply.

[33] A. Hagberg, D. Schult, and P. Swart, "Networkx," Jan. 2015, http://networkx.github.io/.

[34] D. A. da Silva Carvalho *et al.*, "A Graph Reduction Machine for Web Service Compositions," Master's thesis, Universidade Federal do Rio Grande do Norte, 2012 (in Portuguese).

[35] H. B. Medeiros, "PEWS-RT: A runtime system for PEWS," Master's thesis, Universidade Federal do Rio Grande do Norte, 2013 (in Portuguese).

[36] D. Chappell, "Introducing Windows Workflow Foundation," 2012, http://msdn.microsoft.com/pt-br/library/ee210343.aspx.