Automatic run-time versioning for BPEL processes

Paulo Melo, Paulo Rupino da Cunha, Catarina Ferreira da Silva, André Macedo

Noname manuscript No.
(will be inserted by the editor)

Automatic Run-time Versioning for BPEL Processes

Paulo Melo · Paulo Rupino da Cunha · Catarina Ferreira da Silva · André Macedo

the date of receipt and acceptance should be inserted later

Abstract We describe a middleware solution for automatic run-time process versioning in Business Process Execution Language (BPEL) and then analyse its impact in terms of scalability and performance. Business processes change in response to business needs, but the deployment of new versions to a BPEL engine must ensure that running instances are not disrupted and can conclude following their original workflows. Our solution is implemented as a standalone component that manages versioning transparently to the process editor, the orchestration engine, the web services used by the process, and the end-user. We have tested it for almost one year in the production environment of a telecommunications company, without significant overhead in terms of process invocation time.

Keywords Business Process Versioning, Business Process Execution Language (BPEL), BPEL versioning, Service-Oriented Architecture (SOA), Web Services, Middleware

1 Introduction

An increasing number of enterprises have been embracing service-oriented architectures (SOA) and implementing their business processes as orchestrations of web services using BPEL (Business Process Execution Language).

One of the main advantages of this approach is agility. In theory, any changes to the workflows, dictated by new business needs, can easily be made by process owners and then quickly deployed to the execution engine to immediately change system behaviour. However, this is only true in cases where no instances of the modified business process are still running in the engine. Otherwise, the deployment of the new version would leave the running instances in inconsistent states, having partially run the old workflow and seeing it suddenly replaced by a new one. Aborting the running instances before deploying the new version is not a solution, since that would lead to the cancelation of ongoing business process (e.g. consider the installation and activation of a triple-play service at the customer home, with only some of the required steps completed at a given moment). Waiting for existing instances to conclude is often also unfeasible. Several business processes are long-running, meaning that they can take weeks, months, or even years to complete [6, page 23]. Besides, new ones are constantly being started – stopping that would mean stopping the business. Finally, running instances must be allowed to conclude following their original business logic, to ensure that contractual or regulatory issues that applied when the process was started are not violated.

The above constraints mean that an effective SOA implementation needs mechanisms to enable multiple versions of the same process to coexist and run in the engine simultaneously, ensuring that each one follows its original workflow.

Complex as the problem is, several additional challenges emerge when considering the thousands of person-

P. Melo
INESC Coimbra - Instituto de Engenharia de Sistemas e Computadores de Coimbra, R. Sílvio Lima, Pólo II, 3030-290 - Coimbra Portugal and Center for Business and Economics Research (CeBER), University of Coimbra, Portugal, E-mail: pmelo@fe.uc.pt

P. Cunha · A. Macedo
CISUC, Department of Informatics Engineering, University of Coimbra, Portugal

C. Ferreira da Silva
Université Lyon 1, LIRIS, CNRS, UMR5205, F-69621, France

months invested in existing enterprise systems and the stringent controls exerted in production environments. These preclude solutions that require extensive rewriting of code, involve risky changes, degrade performance, or lock-in the enterprise to products that may be discontinued or become inadequate for the workload.

So far, the process versioning problem has been solved by ignoring the latter concern. Several BPEL engines, such as IBM WebSphere Process Server [19] and Oracle BPEL Process Manager [22] (now part of Oracle SOA Suite [23]), implement the required workflow versioning, but in a closed, proprietary manner, not portable across different engines, should the need to replace them arises.

To overcome this situation, Juric et al. have proposed BPEL extensions for versioning, to be used at development, deployment, and run-time [14]. Naturally, this solution requires an engine that supports those extensions and that all existing BPEL processes will be modified to use the suggested syntax. Additional related works are presented in Section 2.

The novelty in our solution is that we propose to handle versioning using only run-time mechanisms, implemented in a middleware component. To match the challenges of the scenario described above, we defined the following design principles:

- High transparency, meaning that the process owner, BPEL editor, BPEL engine, and invoked web services don't need to be aware of our component. The process owner, in particular, can continue to manage his/her own design-time versions of the process source files, without being concerned with the additional automatic versioning that will take place at run-time to account for running instances in new deployments;
- Low invasiveness, avoiding the need to introduce changes to the code of dozens or hundreds of existing of web services and BPEL processes, thus also reducing the risk of introducing new problems in production environments;
- Low overhead, ensuring that the versioning mechanisms do not cause excessive performance penalties;
- High independence, avoiding lock-in to specific BPEL engines or non-standard BPEL syntax extensions;
- High extensibility, enabling the expansion of the middleware component to new uses, such as load balancing, testing and diagnostics, fault injection, or other operations than can benefit from intermediating the exchanges between the BPEL engine and the web services.

In Section 2, we discuss previous work on versioning in SOA environments, after which, in Section 3, we describe the architecture and algorithms for our solution. Section 4 is dedicated to the performance analysis of the proposed solution, based on measurements of almost one year of operation in the production environment of a telecommunications company. Finally, in Section 5, we present conclusions, limitations and future work.

2 Versioning Services and Versioning Long-Running BPEL Processes

Inappropriate support for versioning in SOA has been identified as an important problem [13, 16]. A lot of work has been done on versioning of web services, but much less ground has been covered on the versioning of the BPEL workflows that orchestrate those web services.

2.1 Web Service Versioning

When dealing with versioning of web services due to their evolution, two approaches can be proposed [1]: corrective, changing the producer or the consumer of the service so it continues to interoperate with the new service; and preventive, avoiding changes caused by the evolution on the service that breaks interoperability. Both approaches, however, require some amount of control over either side of the interoperating equation. Service versioning can be achieved explicitly by changing invocations in source code or by using suitably configured UDDI registries, but remains one of the most difficult issues faced by developers of distributed systems [7]. Versioning has also been considered in more specific contexts, including those of service provenance [12, 17] or automatic service matching [5]. In these areas, service versioning may be seen as either something to be natively supported by the environment [17] or achieved using UDDI [12]. In service matching approaches, where more than managing the instanced services directly, the intent is to apply them to new uses, versioning is considered something better handled using the service data description [5], but these approaches do not usually handle processes already running.

The existence of versioning may be mostly transparent to the service user, or it may be exposed in the interface of the services. To support several concurrent versions of the same services, different approaches have been proposed, including creating new XML namespaces or explicit version identifiers [1]. In [10] the difference between implementation versioning and interface versioning was explored, and an infrastructure was proposed that allowed routing of versioned requests to the most recent compatible service (that is, the most recent

one with the same interface version). The proposed implementation, however, was dependent of WebSphere, and did not mention support for asynchronous web services invocation.

A mechanism for transparent version handling of web-services may be the use of proxies to map the requests to the different service versions, while maintaining a fixed service address. Support for proxying web-services can be provided by standardized HTTP proxies, like Squid [25], or by specialized tools geared towards web-services, like the Web Service Gateway support present in WebSphere [21] or the Mule Framework [18].

Fang and colleagues [8] propose an automatic web service versioning support using UDDI and a service interface proxy, but this solution requires the clients to be fitted with specialized proxies to access the service.

In the context of cloud services, TOSCA (Topology and Orchestration Specification for Cloud Applications [20]), an OASIS standard specification, stipulates a language to specify service templates that can define the topology of a service and that can utilize existing process modeling standards to define orchestration (via "plans") that can "invoke the manageability behavior of services" [20]. TOSCA describes what is needed to be preserved across deployments in different environments to enable interoperable deployment of cloud services and their management. Although the TOSCA standard is useful to enable application lifecycle automation while ensuring interoperability [11, 2], it requires rewriting workflows and services in standard-supporting manner. This effort is unfeasible when large scale operations and past investments are at stake.

SOA versioning, however, requires more than versioning web services as described above. It requires versioning of their orchestrations.

2.2 Orchestration versioning

Versioning of orchestrations to support long-running processes is a complex problem for which no completely satisfactory solution is in widespread use. The complexity of the evolution of long-running processes may even require that it itself is considered as long-running [9] where the process migration/evolution is handled on a process-by-process basis, migrating/evolving processes only when they are in a stage where its change will create less disruptions. However, such approach requires inner knowledge on the long-running processes state and also state migration abilities that may not always be present.

Although support for versioning BPEL processes is available internally to some engines, such as IBM WebSphere Process Server [19] and Oracle BPEL Process Manager [22, 23], this approach has some drawbacks. It may be coupled with requirements on the deployment methods used and, being proprietary to each vendor, it is usually not portable across different engines, even if both support versioning. The following alternative approaches have been proposed to endow BPEL processes with versioning:

- Juric and colleagues [14] specify BPEL processes versioning as an integral part of development, deployment, and run-time, by proposing an explicit WS-BPEL extension for versioning. They suggest the use of a version handler to assist in routing requests to appropriate versions of a process and deal with pre-processing and post-processing transformations. They also introduce an extension attribute *bpelx:vid* to specify the required version of the partner link, then enabling the invocation of specific partner link versions. This approach restricts the choice of BPEL engine to those supporting the proposed extensions. To the best of our knowledge, vendors have not implemented these extensions in the years since the paper was published. Should compatible engines exist, legacy BPEL processes would still need to be modified to use the proposed syntax.
- In the context of service choreographies, Baresi et al. [3] distinguish between independent requests, which can use different versions of the same service safely, and dependent requests, which must always exploit the same version. These authors propose extensions to a version consistency algorithm applied to BPEL. They take as input a BPEL process and create a directed acyclic graph which is then labeled to further enable a static analysis of the services' control-flows and determine which are dependent and which are not. Authors deployed their BPEL provisioning and execution environment as an extension of Dynamo, which is an extension of the open-source Active-BPEL engine. In a previous work by the authors, DyBPEL [4] directly addressed orchestration evolution, although not transparently, and did not provide transparent/automatic version management. These authors have also compared their work against quiescence [15], and tranquility [24], that is, against the "classical" approaches for the runtime evolution of component-based distributed systems. However, these approaches can not directly solve the problem of versioning of long-running orchestrations, because these cannot be stopped for the purpose of doing static analysis or be modified at run-time.

The silver lining when dealing with BPEL orchestration evolution is that the interfaces exposed by the BPEL engine may be considered as immutable, so the main challenge of supporting several process versions would be to use the right process version when an orchestration is running, and route the messages to that version by applying a proxying mechanism. Moreover, if the versioning protocol used is kept simple, namely by preventing the reuse of a version that has been replaced – so that the identification of the "current" version and of the "old" versions can be performed automatically – much of the complexity of versioning can be reduced.

In this article we propose a simple framework to handle the process orchestration versioning problem, where neither the producer nor the consumer of the service needs to be modified. This is achieved by using middleware components that handle versioning and proxy the dialog between service provider and consumer. The overall system hides the complexity of versioning from both sides and allows uninterrupted use of the existing software/code base, while supporting an arbitrary number of versions.

3 Design of the Versioning Middleware

Typical work routines should not be disturbed for the high transparency design principle to be met. The user must be allowed to retain full control over his/her own design-time versions of the business processes and deploy them in the same manner. However, since the deployment of a new version of a given process cannot disrupt the instances currently running in the BPEL engine (abort them or leave them in inconsistent states), an additional run-time versioning must occur.

To achieve this, our middleware component intercepts the deployment of a BPEL process from the editor and creates a version with a different name than the one already residing in the engine, thus avoiding overwriting the process model that the running instances are using. The same business process can, thus, originate various run-time versions in the engine, as shown in Figure 1. All but the most current, however, are only kept until the associated running instances terminate. New invocations of the process always use the latest version deployed by the user.

On the left in Figure 1 we see how the various design-time versions of the ActivateDSL process, managed by the user according to business needs, evolve through time. On the right, we see how various run-time versions, managed by our middleware, are kept in the BPEL engine, to ensure that instances of previous process models are not disrupted. Without this run-time versioning, deploying a new user version would replace the one residing in the engine, thus aborting running instances or leaving them in inconsistent states.

The UML sequence diagram in Figure 2 details the algorithm underlying the creation of the run-time versions. As shown there, the deployment of the original BPEL process (e.g. ActivateDSL) from the editor to the engine, is intercepted by the *Deployer* component of our versioning middleware, which, after consulting a repository where we store information about existing run-time versions, determines the next one and adds a sequential Id to the process name before letting the transaction proceed. The answer returned by the engine to the *Deployer* is then passed back to the editor. Neither the BPEL editor nor the BPEL engine are aware of the intermediation made by our *Deployer*, further contributing to the high transparency and low invasiveness design principles stated in the introduction. As a final step, the *Deployer* gets the WSDL used in the invocation of the newly deployed process from the engine and stores it in our versioning repository. It also notifies the *Gateway*, who caches all this information in RAM, all for improved performance.

Having managed to keep multiple run-time versions of the same user process in the engine, then raises the challenge of ensuring that new invocations of a BPEL process and *callbacks* from previously invoked asynchronous web services are routed to the appropriate instances. The algorithm for the simplest case – a new invocation of a process by an external service – is shown in the UML sequence diagram of Figure 3.

New invocations of a process should always use its most up-to-date version (the last one deployed). To this end, upon intercepting an invocation of a process (e.g. ActivateDSL), the *Gateway* determines its most current run-time version (e.g. ActivateDSL_Id), invokes that version, receives its reply, and passes it on to the original external caller.

There is, however, another far more complex scenario: when an external service has been asynchronously called by an instance of a given run-time version of a process and, sometime later, calls back with its reply. In this case, the *Gateway* must ensure that the *callback* is routed to the specific version and instance that made the original call and not, erroneously, to the most recent one. We achieve this by handling BPEL's correlation set – a collection of properties that enable the unambiguous identification of an instance. Correlation sets are already used by BPEL engines in asynchronous exchanges, to ensure that the *callbacks* are directed to the correct caller instance. However, since we created multiple run-time versions for the same user process, we must also intermediate this exchange to ensure that the correct instance of the correct run-time version is iden-

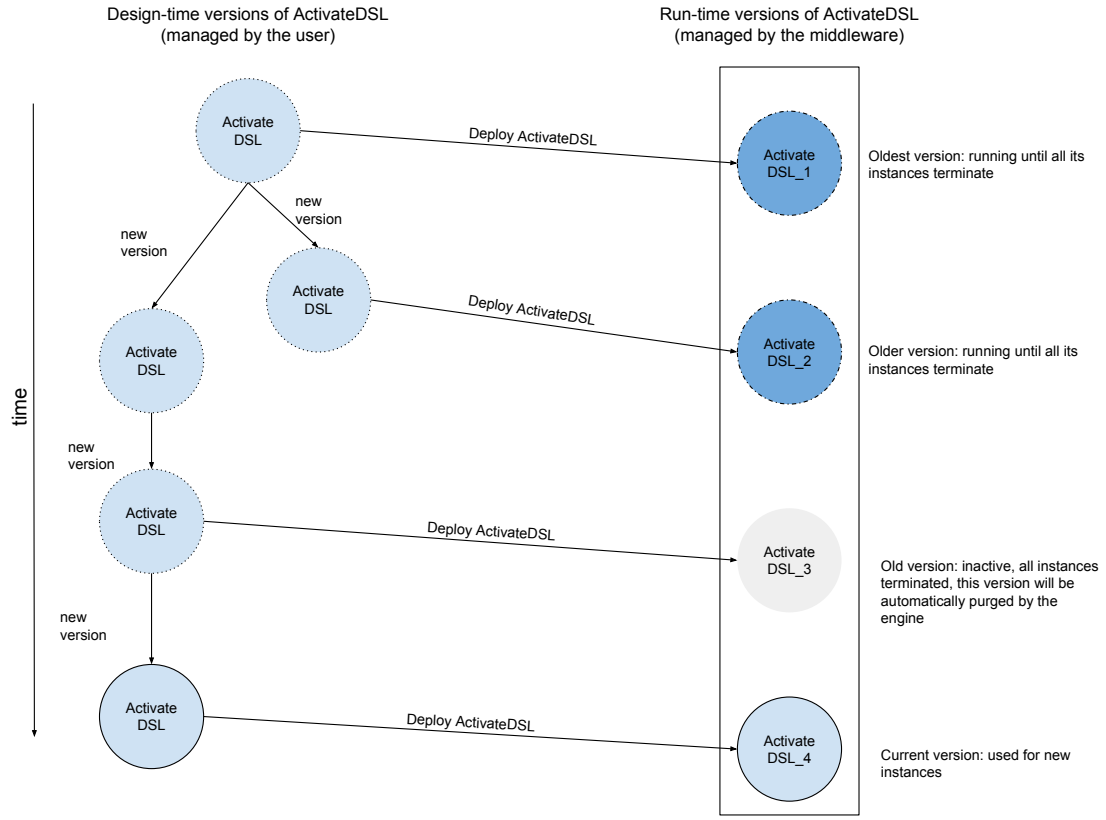

Fig. 1 Mapping of design-time process versions to run-time versions in the BPEL engine

tified. The algorithm to handle *callbacks* originated by asynchronous calls to external services is shown in the UML sequence diagram of Figure 4. As seen there, when the asynchronous callback (1.1) occurs in response to the invocation of the external service (1), our *Gateway* intercepts it. Then, it asks the BPEL engine for all correlation sets for all active instances, finds the run-time instance whose correlation set matches the correlation set of the *callback*, and routes the *callback* to that instance. The reply from the process is passed back to the external service.

In both scenarios – new invocation and asynchronous *callbacks* – neither the external service nor the invoked process residing in the BPEL engine are aware of the intermediation made by our *Gateway*, to ensure the high transparency and low invasiveness design principles. Note how the external service always uses the original process name "ActivateDSL" – thus avoiding the need for modifications to support versioning – leaving it to the *Gateway* to translate as needed to an "ActivateDSL_Id", corresponding to the correct run-time version.

Besides minimizing risky and massive changes to the assets, solutions meant for use in a production environment must not introduce unduly overhead – our third design principle. This issue is particularly relevant for our *Gateway* component, since it intermediates all traffic between external services and processes running in the BPEL engine. Aiming for low overhead, the following architectural decisions were made:

- The WSDLs of the various processes deployed to the engine are stored in a repository of versioning data, for quicker response when requested by a caller. This saves the time it would take for the *Gateway* to call the BPEL engine to get this information. As seen in Figure 2, these WSDLs are stored in the repository by the *Deployer* immediately after it finishes the deployment.
- A cache of these WSDLs and remaining versioning information is kept in RAM by the *Gateway*. Since this component only performs "reads", to find out to which version and instance of a process to forward a message to, a cache speeds up computation significantly. The only "writes" are done by the *De-*

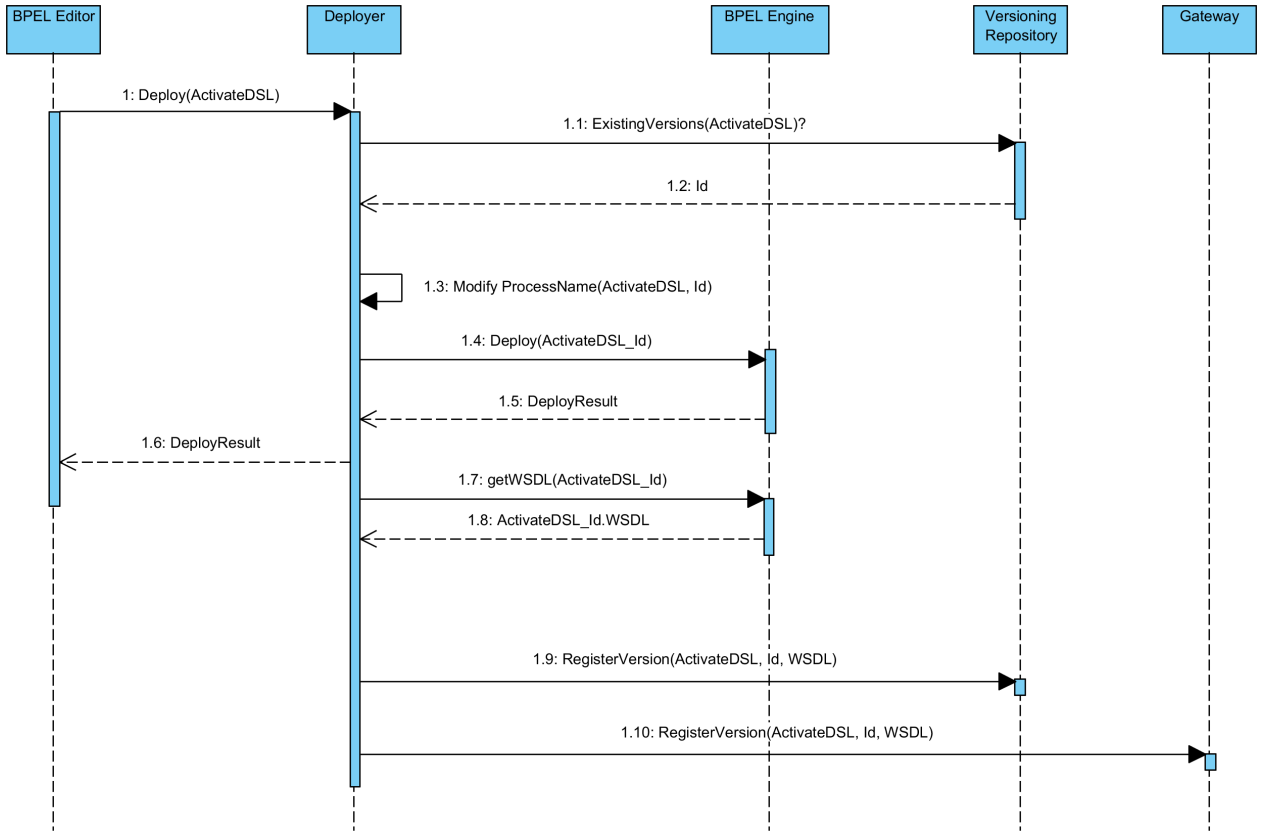

Fig. 2 Deploying a new version to the BPEL engine

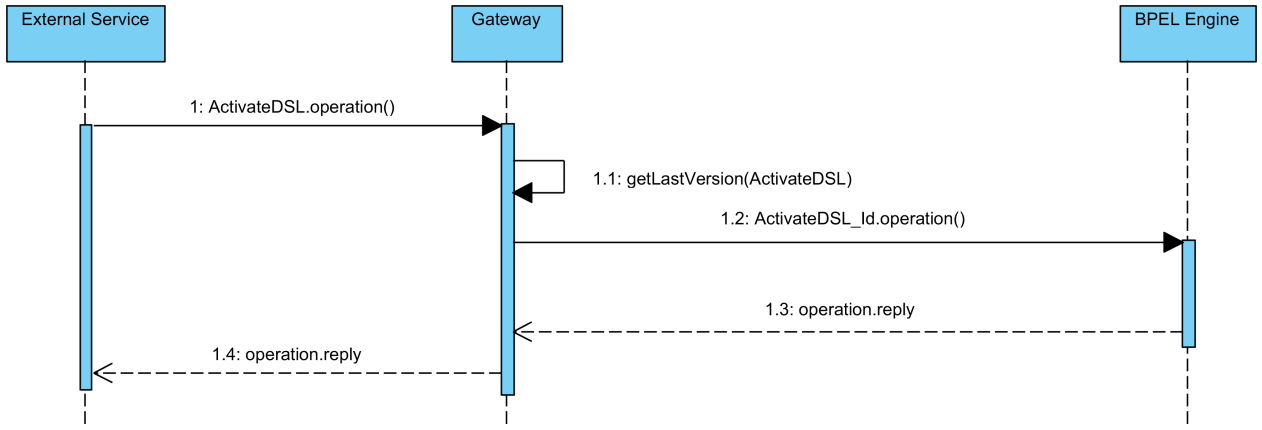

Fig. 3 (New) Invocation of a process by external service

ployer, when it sends a new process to the engine, at which time it also notifies the *Gateway* to update its cache. The versioning repository thus acts just as a persistence layer, to ensure that the whole system can survive reboots.

– The *Gateway* periodically checks the BPEL engine API to verify whether older versions of a given processes still have running instances. If none are found, the *Gateway* will always redirect the invocations to the newest version of the process (saving the call to the API) until a new process is deployed. This optimization, in conjunction with the use of the cache described above, effectively reduces the overhead to almost zero in these cases.

Another relevant aspect of low invasiveness in this architecture is that is does not require modifications to the source code of the BPEL engine, as other solutions do [3]. This is very important, as it enables the adoption of new engine updates as they are released, avoiding the cost and trouble of adapting their source code every time. High independence is also promoted,

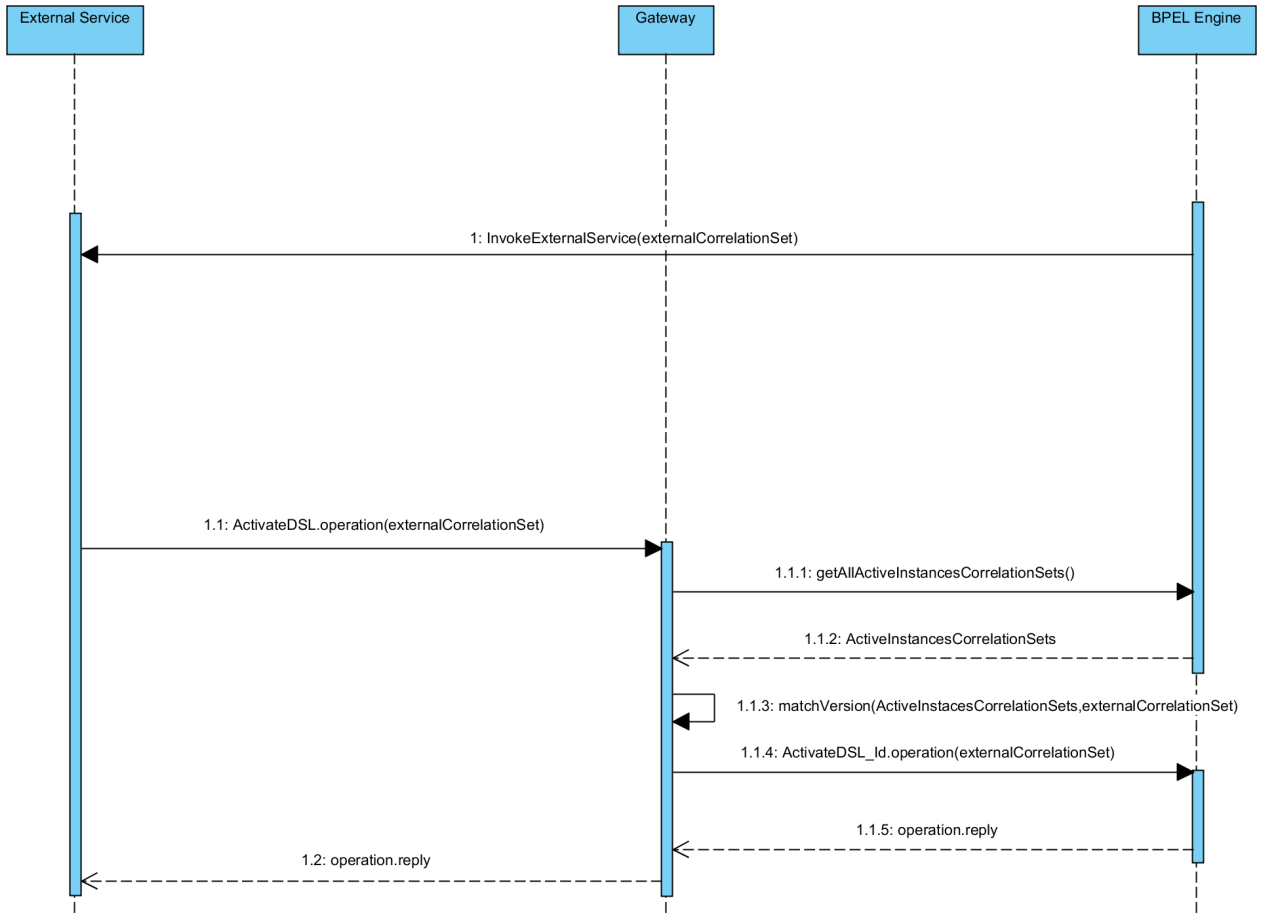

Fig. 4 Assynchronous Invocation of a process using correlation identifiers

since no particular BPEL engine or non-standard BPEL extensions are required. It is highly extensible, since all traffic between external services and processes running on the engine is intercepted and can be manipulated.

In summary, our solution intercepts deployments of business processes to a BPEL engine and creates run-time versions as needed, to maintain the integrity of running instances relying on previous versions of the process model. It then tracks and manages all invocations from external services to the BPEL processes to ensure that:

(a) new ones are routed the most current version;
(b) *callbacks* resulting from assynchronous invocations made by previous versions are routed to the original caller.

Consequently, the user can deploy improved business processes as needed, resting assured that any existing instances that are still running will be allowed to complete following their original workflows. This is of key importance, since several business processes are long-running.

Performance and scalability issues are described in detail in the next section.

4 Field Measurements And Discussion

To measure the overhead created by our versioning middleware we performed some laboratory tests (results shown in Table 1) to evaluate baseline efects on system memory and runtime delays. Those lab tests, ran with various combinations of versions per service and instances per version, shown moderate impact of under 15% increase in memory usage and under 10% increase in process runtime in the worst cases.

Since the previous results were obtained in lab settings, to further validate the approach, data was collected on actual system usage on the production environment of a telecommunications company, for about 200 days distributed throughout almost one year. During this period, the invocation of nearly 2 million processes was recorded. From this data, the following conclusions can be extracted regarding the overhead introduced by our solution.

Most processes experienced very low overhead (less than 20ms in over 98% of invocations) as shown on Table 2. In fact, some kinds of processes never experienced overheads of over 250ms. These processes (called origi-

Table 1 Laboratory Test Results

	Test#	1	2	3	4
Features	Number of versions deployed per service	1	5	5	5
	Number of instances per deployed version	1	1	2	2
	Number of (main) processes	1000	1000	1000	3000
	Total Memory without versioning (MB)	100.448	110.799	115.552	119.325
	Process runtime without versioning (s)	0.397	0.409	0.436	0.457
Memory Increase	Core Memory	7.0%	11.3%	8.0%	9.0%
	Peak Memory	7.9%	13.9%	11.7%	12.2%
	Virtual Memory	6.6%	12.0%	12.1%	8.8%
Runtime Increase	Process runtime	2.2%	7.3%	7.6%	7.0%
	Number of instances created	0.0%	−1.5%	−1.6%	−1.3%
	Number of instances per second	−2.2%	−9.0%	−9.3%	−8.3%

nally "Bus2OM", "PendingManager", "ReSender", "StartProcess" and "XMLManager") are named the "Low Overhead" processes. They were frequently invoked (over 1.9 million invocations in total). Some processes, however, (namely "AutomaticActivity", "ManualActivity" and "Orchestrator") have shown higher average and maximum overheads, although they were less frequent (the logs show about 37200 invocations in total). These processes are therefore called the "High Overhead" ones. It can also be inferred, from the number of versions of those processes present in the system, that they are the longest-running ones, where the additional overhead on invocation will be probably less relevant in the overall process life cycle.

Even for high overhead processes, data shows that almost 90% of invocations presented an overhead of less than 1 second and over 97% of invocations were processed by the versioning middleware in under 2 seconds. This is illustrated in Figure 5, where the data was quantized [26] using bins of 1 second. However some invocations took a lot longer, including a couple of outliers requiring over one minute to process.

A detailed analysis for processes which took more than two seconds (just high overhead processes were found), shows that while the number of high overhead processes in each bin decreases with time, clusters can be found at around 5 seconds and (smaller ones) at 15 and 30 seconds, marked by dashed lines in Figure 5. These clusters may reflect the time required for accessing slower resources. A possibility is that they reflect time required for establishing the original access to the database or the overhead brought by the exhaustion of the database connection pool, both of which would require a time an order of magnitude higher than the simple access to disk, which, in turn, can be an order of magnitude higher than access to memory.

The one big difference between high overhead and low overhead processes, in our telecom case, appeared to be the number of different versions present in the versioning manager at the same time[1], as seen in Table 3. It illustrates, for each process where a certain number of different versions were present in the same day, the average overhead, the average of the maximum overhead and the number of days on which that number of versions occurred. The higher overhead processes are usually those for which more than one version is present. This was expected, since answering calls for processes for which just one version is active should be very fast due to caching. However, there doesn't seem to be a general rule relating average (or even maximum) overhead with the number of processes present in the system (for some processes, the average overhead even seems to decrease with more versions), which may be a sign that higher numbers of versions impose no undue overhead on the versioning system.

To discern whether a lower number of instances in particular days could be responsible for the variance on average overhead according to the number of versions active in the same day, this information is presented on Table 4. It shows, for each high overhead process version with a determined number of versions of that process present in the same day, the number of days in which that situation occurred, the total number of process instances ran, and, for those, the average overhead and average number of instances per day. This data does not seem to support the hypothesis of the number of instances on days with a larger number of versions present being the cause of the variation on the average overhead noticed in Table 3.

To further investigate the relationship between the processes and the overhead recorded, the data was coded into a series and correlated using the Stata™ software.

[1] As the number of concurrent versions of a process present on the versioning manager for each invocation was unavailable, it is assumed to be the number of different versions of the process launched on the same day. While this metric may be incorrect in the case of very long or very short running processes, in this case it was considered a plausible approximation.

Table 2 Process Overhead (Invocations, Average and Maximum – all times in milliseconds)

	Process	Invocations	Average Overhead	Max Overhead
High Overhead	AutomaticActivity	1915	423.015	31665
	ManualActivity	21239	677.513	50385
	Orchestrator	13958	382.862	75735
Low Overhead	BUS2OM	13696	15.002	30
	PendingManager	960739	15.001	210
	ReSender	599911	15.008	90
	StartProcess	361355	15.005	30
	XMLManager	53	15.000	15
	Total	1972866	25.135	75735

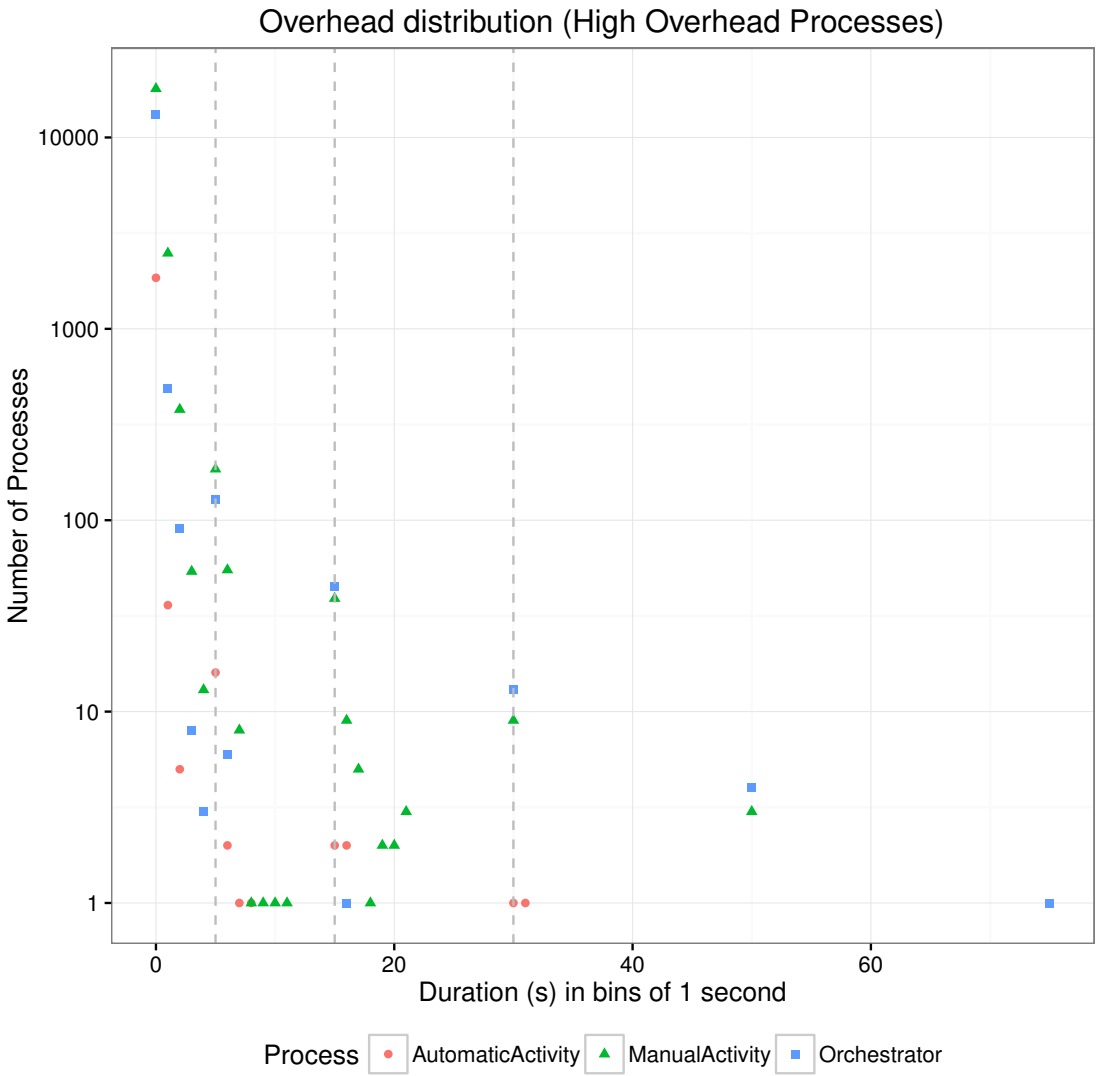

Fig. 5 Overhead Distribution

Table 3 Overhead versus number of versions per day

			# Different Versions Per Day			
			1	**2**	**3**	**Total**
High Overhead	AutomaticActivity	Average Overhead	340.2	626.3		342.4
		AvgMax Overhead	2896.1	626.3		2896.1
		Number of Days	132	1		133.0
	ManualActivity	Average Overhead	624.6	587.3	513.0	609.6
		AvgMax Overhead	2338.4	1487.0	827.0	2338.4
		Number of Days	96	43	5	144.0
	Orchestrator	Average Overhead	343.1	356.5	234.8	345.9
		AvgMax Overhead	1752.8	2858.1	289.4	2858.1
		Number of Days	97	46	2	145.0
Low Overhead	BUS2OM	Average Overhead	15.0			15.0
		AvgMax Overhead	15.1			15.1
		Number of Days	144			144.0
	PendingManager	Average Overhead	15.0			15.0
		AvgMax Overhead	15.0			15.0
		Number of Days	199			199.0
	ReSender	Average Overhead	15.0	15.0		15.0
		AvgMax Overhead	15.0	15.0		15.0
		Number of Days	100	1		101.0
	StartProcess	Average Overhead	15.0	15.0		15.0
		AvgMax Overhead	15.0	15.0		15.0
		Number of Days	168	1		169.0
	XMLManager	Average Overhead	15.0			15.0
		AvgMax Overhead	15.0			15.0
		Number of Days	15			15.0
	Totals	AverageTime	155.1	459.9	433.5	183.7
		MaxTime	2896.1	2858.1	827.0	2896.1
		NumberOfDays	951	92	7	1050.0

Performing an analysis on the correlation[2] between average overhead for the days where both processes shown invocations, it can be shown that the high overhead processes average overheads are correlated among themselves to a high degree, but no such correlation can be found among the low overhead processes or between members of the two groups, as seen on Table 5. The maximum overhead per day also displays the same correlation pattern, as shown in Table 6.

No statistically significant correlation can be found among the number of different versions of each process started per day (except, obviously, the correlation with themselves), as seen on Table 7. This was expected, since the different processes were not necessarily being developed/deployed in sync.

We have also constructed binary variables ("Binary Avg Ov" and "Binary Nr. Versions") which, for each process, codified whether a particular day had higher than average overhead (Table 8) or had a lower than average number of versions present (Table 9). Again, correlation exists among the "usual suspects", that is the high overhead processes, as depicted on Table 8. A weak (-0.3754), even if statistically significant, negative correlation between the low overhead processes "StartProcess" and "PendingManager" is noted, but this may be just a statistical artifact of the coding used, since isn't present in any other correlation.

From the previous correlations it can be suggested that either the high overhead processes are influencing each other or are being influenced in the same way by some environmental factor (e.g. high network latency or high CPU usage on the host computer). It should be noticed that the low overhead processes seem undisturbed by it, which may point to difficulties on the access to resources that aren't required by their versioning.

For all the processes, data shows that the average overhead is strongly correlated with the maximum overhead (and sometimes also with the minimum overhead) but not correlated, with statistical significance, with the number of versions of the processes started each day. Table 10 shows the results for the process "Automatic Activity", but the results were common with small variations among all the processes, both high and low overhead. This seems to imply that, once there is more than one version to be considered, the actual number of versions of a process in versioning has no measurable re-

[2] All correlation tables are shown presenting only the results where a significance level of at least 0.05 (after applying the Dunn-Šidák correction) is found. A dot is shown where not enough valid data was available to ascertain a significant correlation.

Table 4 Average Overhead versus Version number for High Overhead processes according to number of concurrent versions present in the system

Process Name	Version	Versions per day	Days	Instances	Average Overhead	Average Instances per day
AutomaticActivity	0.1	2	1	2	796.50	2
	0.2	1	14	135	347.82	10
	0.3	1	23	298	531.16	13
	0.3	2	1	26	597.96	26
	0.4	1	62	977	488.51	16
	0.5	1	33	477	172.62	14
ManualActivity	0.1	2	19	67	765.37	4
	0.1	3	2	2	820.50	1
	0.2	1	5	177	501.57	35
	0.2	2	14	849	364.90	61
	0.2	3	2	10	909.10	5
	0.3	2	8	12	1510.25	2
	0.3	3	5	87	674.64	17
	0.4	1	11	1222	1008.53	111
	0.4	2	21	1247	1011.76	59
	0.4	3	4	283	769.13	71
	0.6	1	52	9030	738.32	174
	0.6	2	18	2543	522.95	141
	0.6	3	2	245	108.40	123
	0.7	1	28	3301	405.79	118
	0.7	2	6	2164	679.08	361
Orchestrator	0.1	2	17	195	304.40	11
	0.1	3	2	6	304.83	3
	0.2	1	95	9917	297.36	104
	0.2	2	46	1878	920.98	41
	0.2	3	2	52	240.35	26
	0.3	1	2	157	205.99	79
	0.3	2	29	1593	212.58	55
	0.3	3	2	160	192.93	80

Table 5 Average Overhead time-series correlation

	AutomaticActivity	ManualActivity	Orchestrator	BUS2OM	PendingManager	ReSender	StartProcess	XMLManager
AutomaticActivity	1							
ManualActivity	0.5562	1						
Orchestrator	0.7639	0.5472	1					
BUS2OM				1				
PendingManager					1			
ReSender						1		
StartProcess							1	
XMLManager	.	.	.	.	.	.	.	.

Table 6 Maximum Overhead time-series correlation

	AutomaticActivity	ManualActivity	Orchestrator	BUS2OM	PendingManager	ReSender	StartProcess	XMLManager
AutomaticActivity	1							
ManualActivity	0.7623	1						
Orchestrator	0.6005	0.6797	1					
BUS2OM				1				
PendingManager					1			
ReSender						1		
StartProcess							1	
XMLManager	.	.	.	.	.	.	.	.

Table 7 Number of versions per day time-series correlation

	AutomaticActivity	ManualActivity	Orchestrator	BUS2OM	PendingManager	ReSender	StartProcess	XMLManager
AutomaticActivity	1							
ManualActivity		1						
Orchestrator			1					
BUS2OM	.	.	.	.				
PendingManager	.	.	.	.	.			
ReSender	.			.	.	1		
StartProcess					.	.	.	
XMLManager	.	.	.	.	.	.	.	.

Table 8 Correlation among the days with higher than average overhead

	AutomaticActivity	ManualActivity	Orchestrator	BUS2OM	PendingManager	ReSender	StartProcess	XMLManager
AutomaticActivity	1							
ManualActivity	0.5384	1						
Orchestrator	0.5730	0.3085	1					
BUS2OM				1				
PendingManager					1			
ReSender						1		
StartProcess					-0.3745		1	
XMLManager	.	.	.	.	.	.	.	.

lation with the versioning overhead, which leads us to expect good scalability for the versioning solution.

Table 9 Correlation among the days with not less than average number of versions

	AutomaticActivity	ManualActivity	Orchestrator	BUS2OM	PendingManager	ReSender	StartProcess	XMLManager
AutomaticActivity	1							
ManualActivity		1						
Orchestrator			1					
BUS2OM	.	.	.	.				
PendingManager	.	.	.	.	.			
ReSender	.			.	.	1		
StartProcess				.	.	.	1	
XMLManager	.	.	.	.	.	.	.	.

Table 10 Correlation among the different measures for "Automatic Activity"

	Average Ov	Nr. Versions	Maximum Ov	Minimum Ov	Binary Avg Ov	Binary Nr. Versions
Average Ov	1					
Nr. Versions		1				
Maximum Ov	0.8629		1			
Minimum Ov	0.4346			1		
Binary Avg Ov	0.6153		0.2953	0.3803	1	
Binary Nr. Versions		1				1

5 Conclusions and Future work

We presented and discussed the design of a standalone, vendor agnostic middleware, capable of versioning BPEL processes, and then assessed the overhead it introduces under production environment conditions in a telecommunications company. The feasibility of the proposed solution was tested with data recorded for about 2 million process invocations in 200 days distributed across a period of almost one year.

The presented approach to versioning is especially relevant for environments where new instances of processes are constantly being created and those processes are long-running and can take weeks, months, or even years to complete [6, page 23]. Such systems can not be paused, running instances must be allowed to complete following their original workflows, but improvements to the business processes must still be able to be deployed and used for new cases.

Our middleware handles the versioning exclusively via run-time mechanisms, and was architected to be:

- Highly transparent, so that the process owner, the BPEL editor, BPEL engine and orchestrated web services don't need to be aware of its existence. Any high-level versioning done by the user is also unaffected;
- Non-invasive, so that legacy code of dozens or hundreds of web services and BPEL processes don't need to be changed, thus reducing the risk for complex production environments serving millions of customers;
- Low overhead, so that the versioning mechanisms do not become unfeasible in real world environments. Our tests have shown that the impact on performance is negligible, with most processes experiencing a very low overhead, of less than 20ms in over 98% of invocations. Results also show that the overhead does not seem to increase with the number of processes being versioned, thus ensuring good scalability;
- Highly independent, so that the solution does not require a specific brand of BPEL engine or any non-standard extensions to BPEL that might lock-in the company. As designed, the versioning middleware has a limitation in that it requires that whatever BPEL engine is used, it provides a mechanism to inquire about which deployed processes are waiting for external invocations;
- Highly extensible, so that new functionalities can be added. In fact, for future work, we are considering experiments in which the middleware would be modified to act as a load balancer for multiple BPEL engines for high performance scenarios. Other possibilities of expansion include leveraging the intermediation role of the middleware to perform diagnostics and testing in SOA architectures or deliberately injecting faults to assess SOA robustness.

Alternative avenues to pursue in future work include examining the applicability of the approach for non-BPEL services that exhibit the same characteristics (including long runtimes and group identification), and to explore the merits of this approach versus other versioning mechanisms for those services.

Acknowledgements The authors acknowledge the valuable comments and suggestions by the editor and anonymous reviewers.

References

1. V. Andrikopoulos, S. Benbernou, and M. P. Papazoglou. On the Evolution of Services. *IEEE Transactions on Software Engineering*, 38(3):609–628, May 2012.
2. Aria. About ARIA TOSCA. `http://ariatosca.org/about`, Jan. 2017.
3. L. Baresi, S. Guinea, and V. P. L. Manna. Consistent runtime evolution of service-based business processes. In *2014 IEEE/IFIP Conference on Software Architecture*, pages 77–86. Institute of Electrical and Electronics Engineers (IEEE), Apr. 2014.
4. L. Baresi, S. Guinea, and L. Pasquale. Service-oriented dynamic software product lines. *Computer*, 45(10):42–48, Oct. 2012.
5. K. Becker, J. Pruyne, S. Singhal, A. Lopes, and D. Milojičić. Automatic determination of compatibility in evolving services. *International Journal of Web Services Research*, 8(1):21–40, Jan. 2011.
6. M. Brahm, A. N. Fletcher, and H. Pargmann. *Workflow Management with SAP® WebFlow®*. Springer, 2003.
7. K. Brown and M. Ellis. Best practices for Web services versioning. `http://www.ibm.com/developerworks/webservices/library/ws-version/`, Jan. 2004.
8. R. Fang, Y. Chen, L. Fong, L. Lam, D. Frank, C. Vignola, and N. Du. *A Version-aware Approach for Web Service Client Application*, pages 401–409. IEEE, May 2007.
9. D. Frank, L. Fong, and L. Lam. A continuous long running batch orchestration model for workflow instance migration. In *2010 IEEE International Conference on Services Computing*, pages 226–233. Institute of Electrical and Electronics Engineers (IEEE), July 2010.

10. D. Frank, L. Lam, L. Fong, R. Fang, and C. Vignola. *An Approach to Hosting Versioned Web Services*, pages 76–82. IEEE, 2007.
11. S. Harrer, J. Lenhard, G. Wirtz, and T. v. Lessen. Towards uniform BPEL engine management in the cloud. In *Proceedings of the CloudCycle14 Workshop, Stuttgart, Germany, September 22nd 2014*, pages 259–270. Gesellschaft für Informatik, 2014.
12. D. H. A. Ibrahim. *The Concept of Web Service Versioning in Provenance*, pages 469–474. IEEE, Aug. 2009.
13. M. B. Juric and A. Šaša. Version management of BPEL processes in SOA. In *SERVICES, 2010, IEEE 2010 6th World Congress on Services*, pages 146–147, Los Alamitos, CA, USA, July 2010. Institute of Electrical and Electronics Engineers (IEEE).
14. M. B. Juric, A. Šaša, and I. Rozman. WS-BPEL Extensions for Versioning. *Information and Software Technology*, 51(8):1261–1274, Aug. 2009.
15. J. Kramer and J. Magee. The evolving philosophers problem: dynamic change management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.
16. P. Louridas. Orchestrating Web Services with BPEL. *IEEE Software*, 25(2):85–87, 2008.
17. A. Michlmayr, F. Rosenberg, P. Leitner, and S. Dustdar. Selective Service Provenance in the VRESCo Runtime. *International Journal of Web Services Research*, 7(2):65–86, 2010.
18. MuleSoft. Mule user guide: Proxying web services. `https://docs.mulesoft.com/mule-user-guide/v/3.8/proxying-web-services`, 2016.
19. J. Neth, M. Smolny, and C. Zentner. Versioning business processes and human tasks in WebSphere Process Server. `http://www.ibm.com/developerworks/websphere/library/techarticles/0808_smolny/0808_smolny.html`, Aug. 2008.
20. OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA) TC. Topology and orchestration specification for cloud applications version 1.0. OASIS Standard, OASIS, `http://docs.oasis-open.org/tosca/TOSCA/v1.0/os/TOSCA-v1.0-os.pdf`, Nov. 2013.
21. B. O'Hanlon. Create a simple HTTP Web Services Gateway Service with WebSphere Application Server V6. `http://www.ibm.com/developerworks/websphere/library/techarticles/0502_ohanlon/0502_ohanlon.html`, July 2005.
22. Oracle. Oracle BPEL Process Manager Developer's Guide 10g (10.1.3.1.0) Part Number B28981-03. Technical report, Oracle Corporation, 2007.
23. Oracle Corporation. Oracle fusion middleware developing SOA applications with Oracle SOA Suite, 12c (12.2.1.2.0). Technical report, Oracle Corporation, `http://docs.oracle.com/middleware/12212/soasuite/docs.htm`, Nov. 2016.
24. Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, Dec. 2007.
25. D. Wessels. *Squid the Definitive Guide*. O'Reilly Media, Inc, USA, 2004.
26. B. Widrow, I. Kollar, and M.-C. Liu. Statistical theory of quantization. *IEEE Transactions on Instrumentation and Measurement*, 45(2):353–361, Apr. 1996.