Run-time Adaptable Service Oriented Architecture in the Context of Repository Systems

Thiago Gottardi[1], **Rosana T. Vaccare Braga**[2]

[1] Institute of Geosciences and Exact Sciences – São Paulo State University (UNESP)
Av. 24 A, 1515 – 13506-900 – Rio Claro – São Paulo – Brazil

t.gottardi@unesp.br

[2]Institute of Mathematics and Computer Sciences – University of São Paulo (USP)
P.O. Box 668 – 13566-590 – São Carlos – São Paulo – Brazil

rtvb@icmc.usp.br

Abstract. *Service-Oriented Architectures facilitate high modularity and reusability. In this context, service discovery is crucial in identifying new servers that meet the desired client requirements. Although this approach may seem ideal, it is not enough in some situations. For example, in the case of different artifact repositories, they may rely on specific services to adequately adapt to their unique requirements and evolve accordingly. Even considering the availability of common standards for metadata, storing beyond basic metadata and supporting data interchange often requires the development of specific systems. In this paper, we present an architecture for a completely dynamic service-oriented system and use it to implement a prototype of an artifact repository server capable of using different protocols and schemas and reloading new configurations while running. This capability is made possible through an innovative combination of service-oriented architecture, models at run-time, code interpretation, and code generation with just-in-time compiling. This dynamic nature enables the deployment of new configurations on demand without requiring a restart while also facilitating the execution of data processing tasks for various purposes, such as data analysis, in a seamless manner. A working prototype using multiple configurations is used to attest the feasibility of the architecture. We conclude that the solution is viable, but we point to possible scalability and security concerns and how to tackle them.*

1. Introduction

Artifact repositories are good examples of service-oriented architecture systems that are constantly requiring adaptation and evolution. These repositories are developed to facilitate the storage and retrieval of artifacts, including software, scientific studies, data, and metadata. Different standards were proposed to structure the data and metadata of the stored artifacts, as well as how to allow them to become machine-readable. For example, the Reusable Asset Specification[1] (RAS) was published as a schema for metadata of reusable software artifacts.In the context of open data and open science, there are several different specifications, standards, schemas, and metadata formats that are better suited for each case [S Nair and Jeevan 2004]. The standards themselves also face revisions that

[1]http://www.omg.org/spec/RAS/2.2/

depend on a large workload to update the systems and migrate the data. The Digital Curation Centre of The University of Edinburgh maintains a compiled list of several known metadata format standards grouped by disciplines[2].

Many implementations of artifact and metadata repositories provide a programming interface for accessing data via WS (Web Service), which are systems developed to allow the machine to machine communication by using common protocols and formats[3]. However, knowing whether the repository system is actually able to take advantage of the machine-readable principles for metadata depends on each implementation. Therefore, two options are often employed: i) to support a predefined set of metadata formats recommended by the repository to effectively parse their contents according to their machine readability; or ii) to transmit the metadata without marshaling their contents.

In this work, we present a solution that pushes beyond these options through an architecture that we devised to rapidly adapt to the required formats and marshaling needs, without requiring to restart the service. In addition, we added a data processing module capable of running code for the marshaled data for a myriad of applications, e.g., analyses and statistics. We also conclude how each option could be achieved and comment on the required efforts to run and adapt a given WS server at runtime.

The remainder of this paper is structured as follows: In Section 2, we cite and compare related works. Section 3 includes the architecture description of the runtime framework along with use cases. In Section 4, we present the developed prototype that meets the requirements. In Section 5, case studies are presented. In Section 6, we present the final conclusions and plans for future works.

2. Related Work

As our work involves repository systems, it is worth mentioning that there are several consolidated platforms for artifact and data sharing, e.g. Mendeley Data[4] (Elsevier), Zenodo[5] (OpenAire), etc. Even GitHub[6] has been successfully employed as a data-sharing platform. Our architecture was not planned to compete with any of these well-consolidated platforms focused on sharing files and/or tabular data. Indeed, our approach does not involve providing a predefined format for data sharing. Instead, our focus is to address the challenge of adapting to diverse schemas and accommodating various data processing requirements. To accomplish this, we propose a framework incorporating a runtime system capable of rapidly adapting to these schemas and data processing needs.

The earliest examples of dynamic SOA systems using model-driven tools were created by generating J2EE code. Modeling tools allow developers to model the dynamicity of the SOA system before compile-time [Kenzi et al. 2009]. The compile-time requirement encouraged us to focus on the run-time to properly study its challenges and advantages.

In a secondary study describing the state of dynamic SOA with dynamic adaptation targetting Systems-of-Systems, it is discussed that there are several moments when

[2]https://www.dcc.ac.uk/guidance/standards/metadata
[3]http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/
[4]https://www.mendeley.com
[5]https://zenodo.org
[6]https://github.com/topics/datasharing

each task carried by the dynamic system could be performed [Mutanu and Kotonya 2019]. In our work, we focus entirely on the run-time, which poses the biggest challenge among all options. During the development of our approach, we admit that the security concern was not thoroughly explored. Therefore, it is worth citing a security framework for dynamic SOA systems, in which the authors focus on this concern [Kołaczek and Mizera-Pietraszko 2018].

The Arrowhead Approach for SOA Application Development is also a valuable reference of a framework for rapidly constructing service-based applications [Blomstedt et al. 2014]. This approach has been employed in several more recent works, usually focusing on Systems-of-Systems on an industrial context [Kulcsár et al. 2020]. In comparison to our approach, the Arrowhead is a robust framework while our major goal is to present a feasible approach to provide a repository service framework capable of being adapted to different schemas and interfaces at runtime, which was not foreseen by the authors of Arrowhead.

3. The proposed SOA architecture

This paper presents an architecture of a repository system accessible via Web services and a prototype that implements such architecture. Then, we use this prototype in case studies to validate its feasibility. From the perspective of the end user, or the connecting client machine, the prototype behaves just like any other repository system accessible via Web services: client machines submit requests to be responded to by the service server while the interface is well-known and can be shared via service discovery. Although this might seem redundant with existing WS approaches, the novelty value of our architecture relies on the means used to achieve this behavior, as explained in the following.

3.1. Requirements

The proposed architecture involves client and server roles and the information that flows between them. We restrict the roles to *dynamic server with static client* and *dynamic client with static server*. The dynamicity revolves around adapting to new information at run-time. Before run-time, there are three different possible unknown information categories: *unknown schema, unknown interface*, and *unknown processing code*. Therefore, this approach encompasses at least six distinct use cases. It is noteworthy that more combinations are theoretically possible, however, in the case of both clients and servers being dynamic, each information category must be previously known by either peer, which is presented as an extra use case.

Considering both client and server roles and that each of them can be either static or dynamic, any of these may connect to each other. In Table 1, we present the possible connections between clients and servers. We refer to traditional clients and servers as both static, but we are aware that this definition may be confusing since the intent of many services is to provide dynamic responses; in this case, we are referring to the implemented contract: traditionally, clients and servers are implemented according to a predetermined contract. In case the contract changes, the software must undergo modifications. If either the client or the server is dynamic, the dynamic part must adapt to the static part. Finally, when both clients and servers are dynamic, they are expected to implement a contract interpreter.

Table 1. Role Combinations

Part	Static Client	Dynamic Client
Static Server	Traditional SOA System	Client adapts to Server
Dynamic Server	Server adapts to Client	Both implement Model Contract

Table 2. Known and Unknown Information Categories

Part	Information Category	Required-Unknown	Non-Existing
Client	Schema	Ask Server	Blob Client
	Interface	Ask Server	Default Access Proxy
	Processing	Performed by Server	None
Server	Schema	Adapted on demand	Blob Server
	Interface	Adapted on demand	Default Access Services
	Processing	Adapted on demand	Blob Server

Each information category may be known by the part (client or server), unknown or even non-existing. The case of known categories is covered by traditional client-server applications. In Table 2, *Part* indicates whether the part is a client or server; *Information Category* lists the previously described categories; *Required-Unknown* contains descriptions for the cases where the part has this requirement, but this information is unknown at runtime. For example, if the Client requires a Schema but it is unknown, it asks the server; and *Non-Existing* involves the cases where this information is not required at all. For example, if the client has no interface, a default access proxy is used. A further description involving the execution is provided as use cases.

3.1.1. Artifact Types

Three different kinds of artifacts are present in our approach to create a fully dynamic SOA application: WS Definition, Contract Model, and Application code. As illustrated in Figure 1, these artifacts have different forms of representation. Traditionally, the WS Definition may represent the schema and interfaces but not the data processing functions. The advantage of this definition is that existing protocols reference it (e.g., SOAP), allowing the dynamic parts to access the definition even when connecting to static parts. To add further representativity to the definition, we created a new artifact: the contract model. With this model, we can generalize different WS protocols and add custom interfaces for handling concerns specific to dynamic services. It is also supposed to be able to embed application code to be used by the service to process the data. Finally, the application code is the actual implementation of the application, which may be composed of different parts, i.e., client and server. There are four different transformations represented between the artifact types and named as α, β, γ and δ, respectively: from WS Definition to Contract Model; from Contract Model to Application Code; from Application code to Contract Model; from Contract Model to WS Definition. These transformations allow generating artifacts using the contract model as a common language, which is further described in each use case.

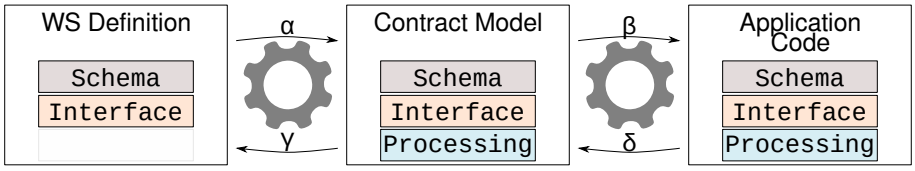

Figure 1. Involved Artifact Types and Transformations

In Table 3, we map the possible artifacts used as a reference for adaptation when the client (or the server) has partial information. We added *None* as a possible category for completeness. If the system requires an information category, the dynamic part should adapt according to the other static or dynamic part. This is made possible by referencing the WS description or the contract model.

Table 3. Artifacts for partial known information categories

Server	Client			
	None	Schema	Interface	Processing
None	traditional	definition	definition	contract
Schema	definition	blob only	definition	contract
Interface	definition	definition	default if.	contract
Processing	contract	contract	contract	no processing

3.1.2. Dynamic Server

The dynamic server responds to service calls according to a predefined WS protocol. Every client request is performed by providing a previously existing service contract model. This contract is represented by a DSM (Domain Specific Model) that is translatable to a service specification, e.g., WADL[7] or WSDL[8] Compared to its counterparts WADL/WSDL, this translatable model has enhanced semantic capabilities as it can effectively represent all information categories encompassing the schema, interface, and processing code. To achieve full dynamicity without requiring new server deployment, the server should adapt to the requested DSM at runtime without creating a new instance.

Although the DSM has the capacity to encompass up to three distinct information categories, it is important to note that these categories are not obligatory. In scenarios where certain information is missing, it is crucial for the server to implement a graceful degradation routine that can compensate for the absence by utilizing default values. However, it is important to acknowledge that relying solely on default values may not be optimal. This requirement was established to simplify the migration process for binary resource repository servers.

In this scenario, each request must be sent to a preexisting service port. If the request header sent by the client has a compatible WS definition, the server must reply accordingly, as expected. However, if it is a non-existing schema, the server assumes that

[7] https://www.w3.org/Submission/wadl/
[8] https://www.w3.org/TR/wsdl/

the data is returned as binary or text, depending on the request headers. If the interface is non-existing, the server assumes that there are basic resource access methods. If the schema or the interface is provided but unknown by the server, the server must perform the α and β transformations to deploy a new port and then redirect the client to a newly created port. Creating a new port lets the client know that the port was created specifically for their request. This avoids an issue we have detected: excessive transparency may cause each client to be oblivious to whether they are using the same service as others. The HTTP protocol, commonly used for implementing WS, already includes response headers that enable the client to receive redirection information. This allows for automatic redirection, if necessary, without causing any inconvenience to end-users. Finally, if the processing code is non-existing, the server assumes that the received data must be untouched.

In case all the information categories are already known, the server must behave according to the contract, and it must reply to clients' requests exactly as a static server would. In addition to the aforementioned scenarios, we introduce the concept of a dynamic client that can request a server to implement a specific dynamic contract. Instead of submitting a traditional request header using a WS Definition reference, the client can provide a contract model to specify its requirements. Therefore, by receiving the contract model, the server would be able to perform the β and γ transformations, making both the new ports available and hosting a WS Definition generated according to the provided contract.

3.1.3. Dynamic Client

A dynamic client must accept contract models at runtime and must be able to invoke services programmatically at runtime according to the contract. Therefore, a dynamic client depends either on the WS definition or the contract model. The end-user or a dynamic endpoint server may provide this, i.e., if the client knows that the endpoint server is also dynamic according to our approach, it should be able to request the contract model. In this case, the client should perform the β transformation to create a service proxy. However, detecting the implemented service by probing a running non-dynamic WS server (or legacy) is not trivial. Despite the availability of the contract model, this model may omit information categories. If the contract does not provide the schema, the client must assume that the server replies using text or binary, according to the contract. If the contract does not provide the interface, the client must assume that the server implements basic resource access methods. If all the information categories are already known, the client will behave exactly as a static client. This is common when the processing code is unknown because it is a server-side definition.

3.1.4. Extra: Both Dynamic

Considering both sides as dynamic, we assume that they are both implemented using our architecture and framework. In this particular scenario, the client has made multiple attempts to locate a suitable server through service discovery but has been unsuccessful. In our framework, we included an extra preliminary handshake sequence executed upon the client's request. The handshake sequence starts with the client submitting the desired

port and the expected contract. If it is feasible to generate a new service based on the requested contract, the server will respond to the client by providing an available port that implements the specified contract. Following the deployment of a new service port, the client may now submit requests similarly to the workflow specified in the previous subsections.

4. Working Prototype

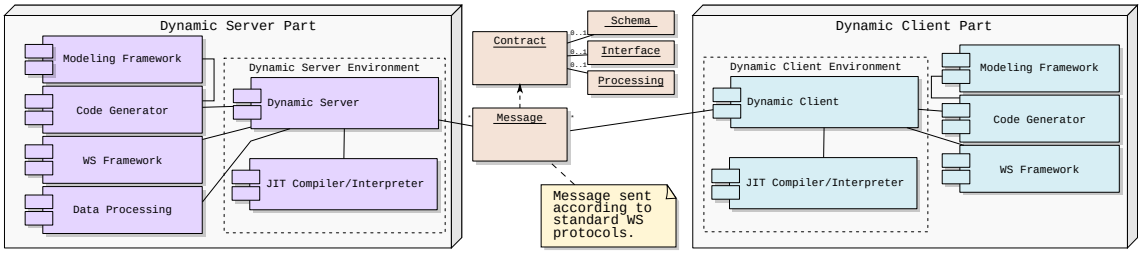

Figure 2. Component diagram for the prototype of the dynamic service-oriented architecture.

For achieving dynamic reconfiguration, our approach combines models at runtime [Bennaceur et al. 2014] with model-driven engineering [France and Rumpe 2007] using just-in-time compilation [Aycock 2003] plus code interpretation [McCarthy 1960].

In Figure 2, a component diagram illustrates the modules that compose our prototype. A message is represented as a class between the server part (left side) and the client part (right side). This message is transferred using standard WS protocols, allowing dynamic parts to connect to standard ones. Whenever a message is sent using a declared schema or style sheet, this is used to identify if a homonym contract is available, otherwise, a new contract model is generated according to the provided schema. The contract is used by both parts as the metadata specification, according to the requirements.

The server part is composed of the dynamic server environment, which contains the service implementation and employs a WS Framework to host service-based connections; and an interpreter capable of running generated code on demand. The code generation is made possible by the *Code Generator* module, which uses the *Modeling Framework* to decode contract models. The *Data Processing* module is specific to the server part. It is a custom module created to manage data on the server side and can invoke data manipulation functions specified by the contract. It has also been used to store data because we have not created a robust data persistence layer for this prototype.

On the client part, the dynamic client environment employs the very same WS Framework, now used to connect to a server. The client works similarly to the server and is composed of a *Code Generator* module, which uses the *Modeling Framework*. However, the generated code needs to be suitable for the client; therefore, the code generation template is different from the server's.

We have implemented a functional prototype that meets the requirements exposed in Section 3.1. While selecting the technologies, the primary goal was to integrate them into a robust existing runtime system that supports all the required libraries. We combined different programming languages and executed it on a JVM (Java Virtual Machine).

For implementing the WS subsystem, we employed Apache CXF[9], because it implements both JAX-WS and JAX-RS. We started by extending an existing dynamic client example using CXF with Java runtime plus its use of Google Guava[10]. Then, to add contract model parsing and code generation, we also had to integrate with model-based code generators, so we selected EMF (Eclipse Modeling Framework) tools. Fortunately, many libraries and frameworks are available in repositories accessible via Apache Maven and Leiningen. However, since their original purpose was different than required by our prototype, we have implemented façade code to allow the parts to invoke the code generator on demand. Then, along with EMF, we were able to include MOFM2T[11] (MOF Model to Text Transformation Language) implemented by Acceleo[12]. MOFM2T is a template-based text generation language that generates source code based on input models. Unfortunately, the requirements for using interpreted code were not adequate to be written purely in Java. In this sense, we elected Clojure[13] as the main programming language for the dynamic parts because its runtime is based on JVM and can compile and execute code on demand, more specifically: deploy web services invoking CXF code, dispatch code generators from Eclipse to generate new code on demand, and finally, run the generated source, compiling just in time whenever possible. The server completely performs this at runtime without restarting.

Another issue had to be dealt with at this point. There are limitations to which code can be executed at runtime: while combining all those technologies might sound simple, generating and running Clojure code at runtime by the same application process required new specific automatic semantic validation and code weaving. In special, it was necessary to replace named class generation using anonymous classes on demand.

5. Case Study

We have conducted case studies with the intent of validating the feasibility of the approach. The case studies include connections and data transmissions between parts (clients and servers), combining dynamic and static parts. These transmissions were carefully planned with the intention of creating realistic data transfers, similar to those carried out by established repository servers that are accessed via WS.

The case studies were executed, and their outcomes were verified both for each module and as a whole, i.e. unit testing and integration testing. All reported studies were executed successfully according to the requirements, as described in Table 4. Notice that the *S*, *I*, and *T* columns indicate whether the schema, interface, or processing code, respectively, are known. Other use cases that were planned for our approach but could not be successfully tested are listed as part of future works.

This study involved executing the prototype according to predetermined use cases. Its major intent was to determine the viability of the overall approach. Different case studies involved mixing both dynamic parts and static parts. It was possible to investigate how the dynamic part was capable to adapt according to new contracts at runtime.

[9] `http://cxf.apache.org/`
[10] `https://guava.dev/`
[11] `http://www.omg.org/spec/MOFM2T/1.0/`
[12] `https://eclipse.org/acceleo`
[13] `https://clojure.org`

Table 4. Successful Test Cases

Description	Client	Server	S	I	P
Traditional WS Test	Static	Static	K	K	K
Server adapts to Client's Schema	Static	Dynamic	U	U	U
Server adapts to Client's WS Definition	Static	Dynamic	K	U	U
Server receives contract, generates and runs code	Dynamic	Dynamic	K	K	U
Client receives anonymous objects and adapts	Dynamic	Static	U	K	U
Client receives contract and generates code to adapt	Dynamic	Dynamic	K	K	K

The result of this work may be affected by some threats to validity, classified according to [Wohlin et al. 2012]. **Internal validity:** There is a possibility that the case studies were not correctly performed according to their requirements. This threat was eliminated by performing unit and integration testing to ensure that the use cases were executed correctly from start to finish. **External validity:** There is a risk that the executed use cases do not represent realistic use cases. Indeed, our prototype has a small scale compared to real repository systems. Yet, we managed to implement functional repository services that would be useful in real-life applications, however, we did not focus on scalability and vulnerability issues that will certainly arise from our approach. We intend to evaluate these issues as part of future works. **Conclusion validity:** The number of contracts and schemas may not be enough to conclude if the prototype is adaptive so as to cover all possible repository formats and interfaces. Perfecting the adaptability of our approach was not the goal of this case study, which should be analyzed in future works. In conclusion, comparative studies could be performed to create a selective guideline for choosing the best option, while our approach could be used to fill the gap of non-consolidated services, redirecting to existing ones when relevant.

6. Conclusion and Future Works

In this paper, we presented an architecture for developing a completely dynamic SOA system. Following these requirements, a prototype was built and then used in case studies. The motivation for creating a dynamic SOA system was to quickly deploy repository systems that adapt to different schemas and interfaces. Moreover, it was possible to push beyond this initial requirement to create completely dynamic SOA systems that adapt to different contracts on demand.

The implemented prototype combines different technologies to generate or transfer contracts, read these contracts, generate code based on contracts and finally, execute the code without restarting the involved parts. The prototype system was composed of two parts: client and server. According to the conducted case studies, the server was successfully used to adapt to clients upon request, regardless of whether the client was also implemented according to our requirements. The client was capable of connecting to the server and redirecting to the new port whenever required.

The conducted case studies involved different scenarios where the client connected to a dynamic server. Our plan is to continue testing the prototype with additional use cases and then provide the corresponding sources [14]. Although the case studies

[14] `https://anonymous.4open.science/r/dynamic-soa`

demonstrate the feasibility of the approach, we acknowledge that this viability may not be realistic or sufficient to address certain scalability issues. We are also aware that several vulnerabilities may arise from running new code on demand for clients. The prototype is still in an early stage of development, and the current studies have primarily focused on the server aspect rather than the client side. Indeed, the server required further efforts to meet the requirements, while the client portion benefited from the preexisting capability provided by Apache CXF and Google Guava. We are working to improve the client part to study how other interpreted languages (e.g. Javascript) could be employed to load dynamic code based on contracts, including support for parsing stored source code, as well as hosting the prototype to implement a well-defined repository system service.

References

Aycock, J. (2003). A brief history of just-in-time. *ACM Comput. Surv.*, 35(2):97–113.

Bennaceur, A., France, R., Tamburrelli, G., Vogel, T., Mosterman, P. J., Cazzola, W., Costa, F. M., Pierantonio, A., Tichy, M., Akşit, M., Emmanuelson, P., Gang, H., Georgantas, N., and Redlich, D. (2014). *Mechanisms for Leveraging Models at Runtime in Self-adaptive Software*, pages 19–46. Springer International Publishing, Cham.

Blomstedt, F., Ferreira, L. L., Klisics, M., Chrysoulas, C., de Soria, I. M., Morin, B., Zabasta, A., Eliasson, J., Johansson, M., and Varga, P. (2014). The arrowhead approach for soa application development and documentation. In *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, pages 2631–2637, Dallas, TX, USA. IEEE.

France, R. and Rumpe, B. (2007). Model-driven development of complex software: A research roadmap. In *2007 Future of Software Engineering*, FOSE '07, pages 37–54, Washington, DC, USA. IEEE Computer Society.

Kenzi, A., Asri, B. E., Nassar, M., and Kriouile, A. (2009). Engineering adaptable service oriented systems: A model driven approach. In *2009 IEEE International Conference on Service-Oriented Computing and Applications (SOCA)*, pages 1–8.

Kołaczek, G. and Mizera-Pietraszko, J. (2018). Security framework for dynamic service-oriented it systems. *Journal of Information and Telecommunication*, 2(4):428–448.

Kulcsár, G., Koltai, K., Tanyi, S., Péceli, B., Horváth, A., Micskei, Z., and Varga, P. (2020). From models to management and back: Towards a system-of-systems engineering toolchain. In *NOMS 2020 - 2020 IEEE/IFIP Network Operations and Management Symposium*, pages 1–6, Budapest, Hungary. IEEE.

McCarthy, J. (1960). Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195.

Mutanu, L. and Kotonya, G. (2019). State of runtime adaptation in service-oriented systems: what, where, when, how and right. *IET Software*, 13(1):14–24.

S Nair, S. and Jeevan, V. (2004). A brief overview of metadata formats. *DESIDOC Bulletin of Information Technology*, 24:11.

Wohlin, C., Runeson, P., Hst, M., Ohlsson, M. C., Regnell, B., and Wessln, A. (2012). *Experimentation in Software Engineering*. Springer Berlin, Heidelberg.