Correct-by-Construction Web Service Architecture

Emmanouela Stachtiari∗, Nikos Vesyropoulos†, George Kourouleas∗, Christos K. Georgiadis†, and Panagiotis Katsaros∗
∗Department of Informatics, Aristotle University of Thessaloniki
54124 Thessaloniki, Greece
Email: {emmastac, gkouroul, katsaros}@csd.auth.gr
†Department of Applied Informatics, University of Macedonia
54006 Thessaloniki, Greece
Email: {nvesyrop, geor}@uom.edu.gr

Abstract—Service-Oriented Computing aims to facilitate development of large-scale applications out of loosely coupled services. The service architecture sets the framework for achieving coherence and interoperability despite service autonomy and the heterogeneity in data representation and protocols. Service-Oriented Architectures are based on standardized service contracts, in order to infuse characteristic properties (stateless interactions, atomicity etc). However, contracts cannot ensure correctness of services if essential operational details are overlooked, as is usually the case. We introduce a modeling framework for the specification of Web Service architectures, in terms of formal operational semantics. Our approach aims to enable rigorous design of Web Services, based on the Behaviour Interaction Priorities (BIP) component framework and the principles of correctness-by-construction. We provide executable BIP models for SOAP-based and RESTful Web Services and for a service architecture with session replication. The architectures are treated as reusable design artifacts that may be composed, such that their characteristic properties are preserved.

Keywords—Service-Oriented Architecture, Rigorous Design, Web Services, Correctness-by-Construction

I. INTRODUCTION

Service-Oriented Computing is a computing paradigm that utilizes services as fundamental units for composing distributed applications. Services are designed as self-described heterogeneous software that may be composed and recomposed into multiple applications and composite services.

The service architecture sets a framework for achieving coherence and interoperability despite service autonomy, heterogeneity in data representation and protocols, and remote system failures. Today’s services adhere to communication agreements called service contracts, which are defined by one or more service description documents and are independent from their implementation. For the sake of the service abstraction principle, we usually lack a detailed formalization of the processing resources for the individual service capabilities. As a consequence, verifying that a service implementation exhibits the expected properties involves intensive testing and debugging in a highly asynchronous and unpredictable environment.

To effectively master the inherent complexity of service design, we would like to be able to establish the correctness of a service, in terms of the way in which it has been composed and deployed, rather than just by testing its operational behavior. “Correctness-by-construction” aims at a design approach with measures that make it difficult to introduce defects and means to detect and remove any defects as early as possible [1]. The role of testing in this case is to validate the design flow and not to find defects.

Correct-by-construction techniques have been successful in hard real-time systems and VLSI design [2]. Regarding software design and in particular services, the necessary ingredients for implementing similar techniques are still an open research problem. Among other obstacles, we currently lack a universally accepted service component model and we already have to cope with a multitude of architectural styles [3]. Additionally, there is no theory with formal design rules that will support reuse of proven service architectures, according to the principles of correctness-by-construction.

In order to reason for the correctness of design artifacts we need a model-driven approach based on a language with formal execution semantics. To cope with the high heterogeneity in the service interaction mechanisms and between the various programming models we need a unified and expressive composition paradigm: coordination between architecture components has to be expressed as architectural constraints specified by composition operators over the components. To this end, we adopt the Behavior Interaction Priorities (BIP) component framework [4], in order to formalize common Web Service architectures and thereafter to enable rigorous design based on the principles of correctness-by-construction.

A formally defined architecture aims to enforce a characteristic property such as session replication, atomicity etc. Architectures are treated as reusable design artifacts that can be composed, such that they do not interfere with each other and their properties are preserved. This perspective of horizontal correctness was first introduced in [2], but we still lack a theory for an adequate composition operator. However, BIP already supports preservation of invariants and deadlock freedom in vertical component refinement. With this design step, we apply action refinement [5] through proven source-to-source transformations [6]. Thus, from a high-level BIP application model, we can derive correct system models that take into account the computational infrastructure and the communication protocols of the execution platform. The ultimate aim is to gradually build correct service implementations through rigorous design.
steps and automatic code generation for the BIP execution engine (standalone or distributed version).

The present article is the second research work towards a complete design flow for correct-by-construction Web Services. In [7], we presented a language embedding for translating WS-BPEL service compositions into BIP application models. Our main contributions in this article are summarized as follows:

- We formalize the operations of a Web Services stack based on a context-free grammar for the logical structure of a BIP model.
- We provide verified executable BIP models for both of the two widely used Web Service architectures, namely WS-* (SOAP-based services) and Representational State Transfer (RESTful services).
- We introduce a formal architecture for session replication [8] and we consider its composition with the WS-* architecture. A fundamental precondition of correctness is deadlock-freedom and non-interference between the two architectures.
- We provide Web Service implementations that run under the control of the BIP execution engine.

In section II, we review the related work and in section III we discuss the current Web Service design practice. The definition of a formal architecture in BIP is provided in section IV, along with a description of the structure of our BIP models for Web Service architectures. In section V, two formally checked models are described, one for SOAP-based and another one for REST service architectures. In section VI, we address the problem of composing a SOAP-based Web Service with an architecture for replicating the session state. Finally, in section VII we discuss two cases of “correct-by-construction” Web Services and we conclude with a critical review of our contribution and the future research prospects.

II. RELATED WORK

A. Service Oriented Architecture and formal methods

Formal methods play an important role in the research of all computing paradigms including Service-Oriented Architectures (SOAs) and their applications. They allow the definition of precise semantics for the languages and protocols of service architectures, and offer a basis for checking the correctness of complex services. We focus on two pervasive Web Service architecture paradigms: SOAP-based and RESTful services.

SOAP-based services are defined in Web Service Definition Language (WSDL) files, which are essentially XML files. They expose a set of operations using two basic interaction patterns: synchronous invocation through remote procedure calls, and asynchronous interactions via message exchange [9]. SOAP-based services represent a procedural style of designing services as they require tightly coupled designs [10]. The SOAP-based approach mainly focuses on the application design and is only secondarily concerned with distribution. Consequently, this service style is preferable in areas that require asynchrony and various service qualities.

On the other hand, RESTful Web Services are primarily concerned with distribution issues [11]. REST represents a navigational style of Web Services and favors a loose coupling of components. Therefore, it is widely acknowledged as a lightweight approach for the provision of services on the Web. Most RESTful Web Services are not described using the standard WSDL language. The REST technology provides a new abstraction for publishing information and giving remote access to application systems: the resource [9]. It uses the inherent expressiveness of HTTP to retrieve representations of Web resources in varying states [12]. The core idea is to support few specific operations (the HTTP methods GET, POST, PUT, and DELETE), for changing the resources’ state.

B. SOAP-based Web Services

Service compositions and the WS-BPEL standard are one of the main areas of research on the use of formal methods to ensure the correctness of applications utilizing SOAP-based Web Services. In this context, the authors of [13] showed that model checking approaches that ignore resource constraints of the deployment environment are insufficient to establish safety and liveness properties of service orchestrations. In order to provide resource-aware process modeling, they used the Finite State Process (FSP) notation [14], a process calculus for concisely describing and reasoning about concurrent programs. Their approach facilitates the analysis, detection and resolution of deadlocks in a Web Services composition. It applies a modeling of Web Service compositions in the form of a translation of BPEL services to FSP and a representation of architectures with resources (such as the distribution of services across hosts, the choice of synchronization primitives in the process and the threading configuration of the servlet container that hosts the orchestrated Web Services).

In [15], the authors address the correctness problem of Web Service protocol implementations by constructing a verifiable Web Service runtime. They proposed a Service-Oriented Description Language for precise and concise description of message processing in Web Service protocol implementations. They also provided an Extended Finite-State Machine model, named FSM4WSR, to formally describe the dynamic behaviors for Web Service protocol implementations.

In [3], the authors focused on the analysis of dynamic reconfigurations in SOAs. They formally defined the service architecture and the relevant mechanisms for service publication, discovery and connectivity as graph transformation systems with type graphs, constraints and transformation rules. After having defined an abstract, business-level architectural style and a refinement relationship between the abstract and the SOA-specific style, they showed how to use this relationship for checking the architecture refinements and how to derive SOA-specific scenarios from given business-level scenarios. They also described how the behavior refinement problem can be formulated as a reachability problem, which can be solved by applying graph transformation and model checking. In the BIP framework, it has been recently introduced support for the
modeling and analysis of dynamic architectures [16]. However, in current work this language extension is not utilized.

C. RESTful Web Services

Developing systems that conform to the principles of the REST architecture is still a challenging task, though the REST technology has become widespread in recent years. Existing frameworks offer implementations of various Web technologies, without providing adequate assistance (e.g. tools for development with the REST guidelines, automated checks for compliance to service profiles etc.) towards a RESTful design flow [17]. Most approaches in related works can be applied to relatively small projects, but for real-world applications a formal approach may be more effective [18]. Recent research on formal models for RESTful systems has been published in [19], [20], [21], [22], [23]. These results indicate that framework developers can use formal models of RESTful systems, if they leverage those models to provide suitable development abstractions that encapsulate the fundamental principles of REST [17].

More specifically, in [19], the authors discussed RESTful process execution on the basis of a special class of Petri Nets. The main concepts of REST, such as intentions at the protocol level and uniform identification of resources, were introduced and mapped to the formal model.

In [20], the authors explored the composition of RESTful services as it is driven by the hypermedia net that is dynamically created while a client interacts with a server. Their proposal is based on Petri Nets, as a mechanism for describing the machine-client navigation.

In [21], the authors introduced a REST metamodel. The metamodel provides a basis for model-driven development, by proposing a vocabulary for REST and a technical perspective that enables modeling of RESTful applications.

In [22], a model for RESTful systems based on a finite state machine formalism is presented. The paper shows how the model enables formalization of the REST architecture constraints, including the uniform service interface, the stateless client-server operation, and the code-on-demand execution.

In [23], the authors used the CSP process algebra to model the REST architecture. The components of a RESTful system are represented by CSP processes and an abstract process algebraic model for each REST architecture constraint is provided. Finally, the authors have used the PAT model checker to verify two of the modeled architecture constraints.

D. Composition and Correctness of Web Service architectures

In a Web Service architecture, an architectural mismatch [24] happens when its components make incompatible assumptions about their interactions or operating environment. Previous works on design-time detection of architectural mismatches ([25], [26]) focus on the declarative specification of architectural styles using the ACME Architecture Description Language. When composing styles, the resulting interaction is checked for possible divergence from the expected message traces. Architectural mismatches can affect fundamental service properties such as responsiveness, reliability and liveness, thus violating correctness of the Web Service architecture.

There is relatively limited research on the correctness and the formal composition of architectures. The main reason may be the lack, until recently, of a truly compositional formalism for the modeling and analysis of complex systems. In light of the latest developments regarding the proof of several attractive properties of BIP (expressiveness, incrementality, scalable analyses and correct-by-construction model transformations), we expect important new results on long-standing research problems related to our work. One such problem is the problem of feature interaction, as it is known in the area of telecommunication systems and services [27]. In the context of our work, this problem appears when a formal Web Service architecture for a SOAP-based or RESTful application is to be composed with another architecture, in order to provide various service qualities such as fault tolerance, atomicity etc.

III. Web Service Design

This section outlines the architecture design decisions in a typical cycle of Web Service development. Currently, a number of frameworks (such as Apache Axis2\(^1\) and gSOAP\(^2\) for SOAP-based services, Django\(^3\), Restlet\(^4\) and MS Casablanca\(^5\) for RESTful services) provide code generation functions for the design and deployment of Web Services. We avoid using a framework-specific terminology and we focus on the fundamental decisions in the design of a Web Service.

A. SOAP-based Web Services

- Application functionality is implemented in the form of software components that initially are not bound to any service context.
- A number of software components contributing to a business goal is grouped into a module that will be deployed as a service with one or more bindings. Each binding defines the transport protocol to be used and the message style ("RPC" or "document").
- For each binding and each component to be exposed as a service task, a soapAction HTTP header and the message encoding style are defined. The soapAction header is attached to the incoming service requests, in order to simplify server-side processing. A message encoding style such as SOAP encoding assigns a set of rules for serializing data in the body of SOAP messages.
- The framework generated code is eventually deployed together with the service on an HTTP server.
- The WSDL service description can be published under a single portType (set of abstract operations).

1\(\)http://axis.apache.org/axis2/  
2\(\)http://gsoap2.sourceforge.net/  
3\(\)https://www.djangoproject.com/  
4\(\)http://restlet.com/  
5\(\)http://casablanca.codeplex.com/
B. RESTful Web Services

The REST architecture aims at the creation of lightweight, resource-based services. These services are characterized by reduced complexity compared to SOAP-based services, since the need for exchanging complex SOAP messages is eliminated. In REST, Web Services communicate through a uniform interface based on the HTTP methods (there is no need for explicit binding definition).

- Data to be exposed as resources are identified and may be accessed through URIs that are assigned to them.
- For each resource, appropriate HTTP methods (e.g., GET, POST etc.) are selected to manage the corresponding data and the media types supported as alternative resource representations (e.g., XML, JSON, ATOM etc.) are defined.
- Application functionality is implemented in the form of software components that will be exposed as service tasks. Resource URIs and the HTTP methods used to access them are called from dedicated program methods. Every task consumes the provided resources, modifies them and may return hyperlinks to allow invocation of other tasks.
- The framework generated code integrates the implemented tasks that are deployed as a service on a REST-compliant HTTP server.

IV. Formal Web Service Architecture

In this section, we provide an introduction to the BIP component framework and a formal definition of what constitutes a formal architecture. Also, we describe the structure of the BIP models that represent Web Service architectures.

A. Introduction to the BIP component framework

BIP is an expressive component framework with rigorous semantics. It allows the construction of complex, hierarchically structured models from atomic components characterized by their behaviour and their interfaces. We provide the formal semantics of the BIP framework, as is defined in [28] and earlier articles.

BIP atomic components are transition systems extended with a set of ports and a set of variables. An atomic component \( C \) is formally defined as a tuple \( (Q, X, P, T) \), where \( Q \) is a set of control locations, \( X \) is a set of variables, \( P \) is a set of communication ports and \( T \) is a set of transitions. Each transition \( \tau \) is of the form \( (q, p, g, f, q') \) where \( q, q' \in Q \) are control locations, \( p \in P \) is a port, \( g \) is a guard and \( f \) is the update function of \( \tau \). \( g \) is a predicate defined over variables in \( X \) and \( f \) is a function (BIP can invoke functions written in C/C++) that computes new values for \( X \) according to their current values.

In order to compose a set of \( n \) atomic components \( \{C_i = (Q_i, X_i, P_i, T_i)\}_{i=1}^{n} \), we assume that their respective sets of ports and variables are pairwise disjoint. We define the global set \( P \triangleq \bigcup_{i=1}^{n} P_i \) of ports. An interaction \( a \) is a triple \( (P_a, G_a, F_a) \), where \( P_a \subseteq P \) is a set of ports, \( G_a \) is a guard and \( F_a \) is a data transfer function. By definition \( P_a \) contains at most one port from each component. We denote \( P_a = \{p_i\}_{i \in I} \) with \( I \subseteq \{1, \ldots, n\} \) and \( p_i \in P ; G_a \) and \( F_a \) are defined on the variables of participating components, that is \( \bigcup_{i \in I} X_i \).

Given a set \( \gamma \) of interactions, a priority is defined as a strict partial order \( \pi \subseteq \gamma \times \gamma \). We write \( a \pi b \) for \( (a, b) \in \pi \), to express the fact that interaction \( a \) has lower priority than \( b \).

A composite component \( \pi\gamma(C_1, \ldots, C_n) \) is defined by a set of components \( C_1, \ldots, C_n \), composed by a set of interactions \( \gamma \) and a priority \( \pi \subseteq \gamma \times \gamma \). If \( \pi \) is the empty relation, then we may omit \( \pi \) and simply write \( \gamma(C_1, \ldots, C_n) \). A global state of \( \pi\gamma(C_1, \ldots, C_n) \) where \( C_i = (Q_i, X_i, P_i, T_i) \) is defined by a pair \( (q, v) \), where \( q = (q_1, \ldots, q_n) \) is a tuple of control locations such that \( q_i \in Q_i \) and \( v = (v_1, \ldots, v_n) \) is a tuple of valuations of variables such that \( v_i \in Val(X_i) = \{\sigma : X_i \rightarrow D\} \), for all \( i = 1, \ldots, n \) and for \( D \) being some universal domain.

The behaviour of a composite component without priority \( C = \gamma(C_1, \ldots, C_n) \) is defined as a labelled transition system over the set \( S \) of global states of \( C \) and the transition relation with the following semantics: \( C \) can execute an interaction \( a \in \gamma \) iff (i) for each port \( p_i \in P_a \), the corresponding atomic component \( C_i \) allows a transition from the current location labelled by \( p_i \) (i.e. the corresponding guard \( g_i \) evaluates to true), and (ii) the guard \( G_a \) of the interaction evaluates to true. If these two conditions hold true for an interaction \( a \) at state \( (q, v) \), then \( a \) is enabled at that state. Execution of \( a \) modifies participating components’ variables by first applying the data transfer function \( F_a \) on variables of all interacting components and then the update function \( f_i \) for each interacting component. The local states of components that do not participate in the interaction stay unchanged.
In the BIP language, interactions between components are specified by connectors. A connector defines a set of interactions based on the synchronization attributes of the connected ports (Fig. 1i), which may be either trigger or synchronous:

- if all connected ports are synchrons, then synchronization is by rendezvous, i.e., the defined interaction may be executed only if all the connected components allow the transitions of those ports (Fig. 1ii),
- if a connector has one trigger, the synchronization is by broadcast, i.e., the possible interactions are all non-empty subsets of the connected ports that contain the trigger port (Fig. 1iii).

Connectors can export their ports for building hierarchies of connectors (Fig. 1iii). Furthermore, a port may be used to compute transfer functions associated with interactions.

B. Formal architecture

Formally specified architectures in BIP are the means for applying two fundamental principles for the construction of composite components that meet certain properties [2]: property enforcement and property composability. The enforcement of a given property over a set of components is done by restricting their behavior, so that the resulting behavior meets that property. Formal architectures are seen as solutions to a coordination problem, in order to ensure a characteristic property that assigns them a meaning that can be understood without the need for explicit formalization (e.g., scheduling policy). Property composability deals with the problem of combining more than one architectures on a set of components, in order to achieve a global property. This section is concerned with the definition of formal architectures. The discussion for the composition of architectures is postponed for section VI-A.

An architecture is defined in terms of a glue operator, say \( gl \), used to specify a particular set of interactions between the coordinated components \( C_1, \ldots, C_n \). The partial function of the transition relations of the components. More specifically, if \( C_i \) can execute in states \( s_i \) transitions of the form \( s_i \xrightarrow{a} s_i' \), then \( gl(C_1, \ldots, C_n) \) can execute transitions of the form \( (s_1, \ldots, s_n) \xrightarrow{a} (s_1', \ldots, s_n') \) where \( a \) is an interaction, i.e., a non-empty subset of \( \{a_1, \ldots, a_n\} \) such that \( s_i'' = s_i' \) if \( a_i \in a \) and \( s_i'' = s_i \) otherwise.

**Definition 1** ([2]). An architecture is a context \( A(n)[X] = gl(n)(X, D(n)) \), where \( gl(n) \) is a glue operator and \( D(n) \) a set of coordinating components, with a characteristic property \( P(n) \), parameterized by an integer \( n \) such that:

- \( A(n) \) transforms a set of components \( C_1, \ldots, C_n \) into a composite component \( A(n)[C_1, \ldots, C_n] = gl(n)(C_1, \ldots, C_n, D(n)) \), by preserving essential properties of the composed components, that is,
  1. **Deadlock-freedom:** if components \( C_i \) are deadlock-free then \( A(n)[C_1, \ldots, C_n] \) is deadlock-free too;
  2. **Invariants (state predicates preserved by the transition relation):** any invariant of a component \( C_i \) is also an invariant of \( A(n)[C_1, \ldots, C_n] \).

\( A(n)[C_1, \ldots, C_n] \) meets the characteristic property \( P(n) \).

It has been proved that composition by glue operators preserves the invariants of their arguments [29].

C. The Web Service architecture \( A_{WS} \)

\( A_{WS} \) provides the general structure of Web Service architectures in terms of a grammar with all necessary types of behavior:

\[
\begin{align*}
\langle \text{ws_arch} \rangle & ::= \langle \text{srvc_c} \rangle + \langle \text{srvc_s} \rangle + \langle \text{srvc_cs} \rangle * \text{ network} \\
\langle \text{srvc_c} \rangle & ::= \text{invc_endpt+ task_c+ data*} \\
\langle \text{srvc_s} \rangle & ::= \text{list dispch+ task_s+ data*} \\
\langle \text{srvc_cs} \rangle & ::= \text{list dispch+ invc_endpt+ task_cs+ data*}
\end{align*}
\]

The BIP model \( \text{ws_arch} \) for \( A_{WS} \) consists of services acting as clients (\( \text{srvc_c} \)), servers (\( \text{srvc_s} \)), or both roles (\( \text{srvc_cs} \)). Services communicate through the Internet (network).

A \( \text{srvc_s} \) consists of a number of service tasks (\( \text{task_s} \)) and maintains a listener (\( \text{listen} \)) for receiving connection requests for the destination addresses (IP address and port) of the tasks. Also, \( \text{srvc_s} \) employs dispatchers (\( \text{dispch} \)) which process incoming requests and route them to the requested \( \text{task_s} \). There must be one dispatcher for every group of tasks that share the same combination of destination address and the (message style, message encoding) binding elements. Servers may utilize data components for e.g., maintaining the application state shared by the \( \text{task_s} \) components.

A \( \text{srvc_c} \) includes a number of application tasks (\( \text{task_c} \)) for initiating invocations of multiple \( \text{srvc_s} \). A \( \text{srvc_c} \) creates an invocation endpoint (\( \text{invc_endpt} \)) for each \( \text{srvc_s} \) interface specification (e.g., WSDL file), in order to prepare and send the invocation request messages. Clients may also store information (e.g., session IDs) in data components.

The behavior of \( \text{invc_endpt} \) and \( \text{dispch} \) components encompasses the functions of message (de-)serialization and transfer, which vary with the transport protocol and the transformation between the messages’ format and a representation that can be processed by the tasks. Additional functions that may be offered by these components are user authentication and logging.

In \( A_{WS} \), the components export the ports which are shown in Fig. 3 and are coordinated with the connectors depicted in Fig. 2. These ports constitute the required interface for component coordination in every Web Service architecture. Component behaviors do not have to enable all ports.

---

*Exporting a union of ports is a new feature introduced in BIP 2.0.*
The data component receives through the query port a certain query to execute (qry data) and returns the result (rslt) through the result port. The components can submit queries (sndQry) to data and get their results (getRslt).

The task_c initiates an invocation (out Req) using a destination address (dest), an input (inp), a binding (bind) and optional settings (opt) such as a time-out deadline and gets (in Resp) the invocation’s result (outp). On the other hand, the task_s is invoked (in Req) with some input and returns (out Resp) the result.

The invc_empt gets (prep Req) from task_c the input, binding and destination of an invocation to be made. It attempts to connect (init Conn) to the server and is notified upon the establishment (conn Acc) or rejection (conn Ref) of a connection (conn). Also, it sends (snd Req) requests (req), receives (rcv Resp) responses (resp) and delivers (dlvr Resp) the output of invocations to task_c.

The listn accepts (accept) or rejects (reject) a connection with a srvc_s and dispatches a connection to a dispch. The dispch is initiated (init) for a given connection, receives (rcv Req) invocations and delivers (dlvr Resp) to a task the invocation’s input. Also, it gets (prep Resp) the result from a task in order to prepare and send (snd Resp) a response (resp).

V. SOAP-BASED AND REST ARCHITECTURES

We introduce instantiations of A_Ws, for the two common Web Service architectures, namely the SOAP-based (A_SOAP) and the REST (A_REST). Both A_Ws instantiations include the same behavior types with the ones shown in the grammar of section IV-C with the same component interfaces and connectors. A_SOAP and A_REST specify different architecture constraints by varying the components’ behavior and the used data types. Consequently, the glue operators gl_SOAP and gl_REST (cf. definition 1) are parameterized as appropriate.

A. Correct-by-construction Web Service design

Architectures A_SOAP and A_REST are initial design artifacts, which may be composed with architectures that enforce additional characteristic properties for the development of correct-by-construction Web Services. This takes place in a two-phase design flow: (i) in the model-based design phase, correctness is the primary concern, (ii) in service implementation, the model is transformed into an executable program. During model-based design, a prerequisite for establishing correctness is that the srvc_s component activates all possible srvc_s execution scenarios independently of the messages that are sent to srvc_s. During service implementation, we...
need to attach to the model appropriate code for processing the valuations $v_i$ of the model variables derived from the concrete client messages. In more detail, the proposed design flow is outlined as follows:

- **Model-based design**
  - The $\text{task}_\text{s}$ components are checked with the BIP tools for application-specific properties and deadlock-freedom.
  - We select appropriate binding(s) for the service tasks to a Web Service architecture and we add one $\text{disptch}_{\text{s}}$ component for each binding. The new components are also deadlock-free.
  - The $\text{srvc}_\text{c}$ component for a client is created, with one $\text{task}_\text{c}$ for every $\text{task}_\text{s}$ to be invoked and one $\text{invc}_{\text{endpt}}$ for every binding used.
  - The components are composed with the glue $\text{gl}_{\text{SOAP}}$ or $\text{gl}_{\text{REST}}$, such that deadlock-freedom and component invariants are preserved, as foreseen by Def. 1. This is verified with the existing BIP tools that apply appropriate checks on the model structure. At this stage, the content of the exchanged messages is irrelevant, since we only have to assign values to the model variables such that all possible $\text{srvc}_\text{c}$ execution scenarios are activated.

- **Service implementation**
  1. All connectors with interactions between $\text{srvc}_\text{c}$ and network are removed. The ports used in these connectors will not be exported by the $\text{srvc}_\text{c}$ interface.
  2. We attach C++ code for processing the valuations $v_i$ of the model variables, which will be invoked upon the execution of transitions within the $\text{disptch}, \text{task}_\text{s}, \text{listen}$ and data components.
  3. We attach C++ code for communicating with network sockets to the $\text{disptch}$ and $\text{listen}$ components.

### B. SOAP-based Web Services

The SOAP protocol allows Web Services to exchange structured information. SOAP works over a transport protocol that in most cases is the HTTP. In our model for $A_{\text{SOAP}}$ we adopt the assumption that the transport protocol used is HTTP.

A SOAP message is wrapped in an envelope XML element, which contains an optional $\text{header}$ and a required $\text{body}$ with an optional $\text{fault}$ section. Non XML data can be also attached.

The header section informs the recipient about application-specific functions that are described in the WS-* specifications. It is mandatory for the recipient to recognize and process the elements of header which have the $\text{mustUnderstand}$ attribute. The body section includes the message payload that is intended for the recipient service task. The fault section indicates a SOAP error, which may happen during message processing. Every SOAP message is preceded by the initial line of an HTTP request or response, which contains the destination address (URI) or the HTTP status of the request’s result, respectively. The initial line is followed by HTTP headers with various operating parameters like the content type, the length of the transmitted message, routing information etc.

$A_{\text{SOAP}}$ aims to enforce the characteristic property:

$$P_{\text{SOAP}} : \text{contract-based service communication}$$
Essential architecture constraints for enforcing $P_{\text{SOAP}}$ are: (1) $\text{invc}_\text{endpt}$ structures service requests and parses responses according to an a-priori known WSDL definition, (2) $\text{invc}_\text{endpt}$ and $\text{disptch}$ handle possible message parsing errors, if the processed messages do not comply to the expected WSDL definition.

The aforementioned constraints are implemented by the $\text{invc}_\text{endpt}_S$ and $\text{disptch}_S$ components (shown in Fig. 4) that correspond to the $\text{invc}_\text{endpt}$ and $\text{disptch}$ behavior types of $A_{\text{WS}}$. We define $A_{\text{SOAP}}$ as in Def. 1 by enclosing the constituents of the composite components in curly brackets:

$$A_{\text{SOAP}} = \{ \text{srvc}_c\{\text{task}_c^+ \text{ data}^+ \text{ invc}_\text{endpt}_S^+\}, \text{srvc}_s\{\text{task}_s^+ \text{ listn} \text{ data}^+ \text{ disptch}_S^+\} \text{, network} \}$$

Essential architecture constraints for enforcing $P_{\text{SOAP}}$ are: (1) $\text{invc}_\text{endpt}$ structures service requests and parses responses according to an a-priori known WSDL definition, (2) $\text{invc}_\text{endpt}$ and $\text{disptch}$ handle possible message parsing errors, if the processed messages do not comply to the expected WSDL definition.

The aforementioned constraints are implemented by the $\text{invc}_\text{endpt}_S$ and $\text{disptch}_S$ components (shown in Fig. 4) that correspond to the $\text{invc}_\text{endpt}$ and $\text{disptch}$ behavior types of $A_{\text{WS}}$. We define $A_{\text{SOAP}}$ as in Def. 1 by enclosing the constituents of the composite components in curly brackets:

$$A_{\text{SOAP}} = \{ \text{srvc}_c\{\text{task}_c^+ \text{ data}^+ \text{ invc}_\text{endpt}_S^+\}, \text{srvc}_s\{\text{task}_s^+ \text{ listn} \text{ data}^+ \text{ disptch}_S^+\} \text{, network} \}$$

Essential architecture constraints for enforcing $P_{\text{SOAP}}$ are: (1) $\text{invc}_\text{endpt}$ structures service requests and parses responses according to an a-priori known WSDL definition, (2) $\text{invc}_\text{endpt}$ and $\text{disptch}$ handle possible message parsing errors, if the processed messages do not comply to the expected WSDL definition.

Fig. 4 shows the transitions for the atomic components $\text{invc}_\text{endpt}_S$ and $\text{disptch}_S$ by highlighting a number of internal transitions used to implement the discussed architecture constraints. In Fig. 5 we describe the semantics of these transitions, while the semantics of all other transitions can be found in Fig. 3. We note that the $\text{tick}$ ports represent clock ticks which allow the components to leave a state when a time-out condition is met.

Architecture $A_{\text{SOAP}}$ has been checked for deadlock-freedom with the BIP verification tools.

Fig. 6: The behaviour and interactions that implement the constraints of $A_{\text{REST}}$.

Essential architecture constraints for enforcing $P_{\text{REST}}$ are: (1) $\text{srvc}_s$ includes in responses the session state of $\text{srvc}_c$, (2) $\text{srvc}_c$ includes the session state in request messages sent to $\text{srvc}_s$. As a consequence of the aforementioned constraints, there is no need for $\text{srvc}_s$ to maintain information for communication with individual clients.

$P_{\text{REST}}$ is enforced by the following architecture constraints: (1) $\text{srvc}_s$ includes URIs in the response messages (2) the content of the next request message to be send to $\text{srvc}_s$ is selected by $\text{task}_c$ from the previously received URIs.

The architecture constraints for $P_{\text{REST}}$ instantiate the $\text{data}$ behavior type to represent the cache memory (component cache). These constraints are: (1) $\text{invc}_\text{endpt}$ retrieves the version number of the cache entry associated with a requested URI and attaches it to the message to be sent, (2) $\text{disptch}$ checks in the local cache if the version of $\text{srvc}_c$ is fresh, (3) $\text{disptch}$ delivers the request to $\text{task}_s$ only if the requested URI is not found in local cache, (4) $\text{disptch}$ attaches to the responses an appropriate cache control header.

**C. RESTful Web Services**

Service request messages in REST consist of (i) a URI used to access an exposed resource, (ii) an HTTP method and (iii) HTTP headers expressing communication directives such as the resource representation media type in the expected response, cache control directives etc. Requests might also contain additional information in some format, such as plain text or XML. Response messages deliver a status code and structured documents representing the resources’ current state.

$A_{\text{REST}}$ aims to enforce the characteristic property:

$$P_{\text{REST}} : P_1 \wedge P_2 \wedge P_3$$

where

- $P_1$: services are stateless
- $P_2$: services expose a uniform interface
- $P_3$: service output can be cached
The constraints are implemented by the $invC_{endptR}$, $dispatchR$ and $cacheR$ components (shown in Fig. 6) that correspond to the $invC_{endpt}$, $dispatch$ and $data$ behaviour types. We define $A_{REST}$ as follows:

$$A_{REST} = \begin{cases} 
\text{s}_c & (\text{task}_c \times \text{data} \times \text{invC}_{endptR} \times \text{cache}, \text{s}_r \text{task}_s \times \text{listn} \times \text{data} \times \text{dispatchR} \times \text{cache}, \text{network}) \\
= \text{g}_{REST}( \\
\text{s}_c & (\text{task}_c \times \text{data} \times \text{invC}_{endptR} \times \text{cache}, \\
\text{s}_r & \text{task}_s \times \text{listn} \times \text{data} \times \text{dispatchR} \times \text{cache}, \\
\text{network}) 
\end{cases}$$

In Fig. 6, we highlight the key transitions used to implement the discussed architecture constraints, whose semantics are described in Fig. 7. Architecture $A_{REST}$ has been checked for deadlock freedom with the BIP verification tools.

<table>
<thead>
<tr>
<th>Port</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>invC endptR</td>
<td>Prepares a request out of a URI and possibly a given input, the session state information and the version number of the resource in cache, if one is found.</td>
</tr>
<tr>
<td>buildReq</td>
<td>Parses the response and transforms the returned URIs and session state to the input format for task_c.</td>
</tr>
<tr>
<td>parseResp</td>
<td>Skips caching, if the response cannot be cached or if a failure status has been returned (e.g. 404 Not Found).</td>
</tr>
<tr>
<td>dispatchR</td>
<td>Identifies the requested task_s by extracting the HTTP method and the URI. Error occurs if the request cannot be matched to some task_s.</td>
</tr>
<tr>
<td>matchTask</td>
<td>Client’s or server’s cached result is fresh.</td>
</tr>
<tr>
<td>isFresh</td>
<td>Parses the request and transforms it into the input format for task_s.</td>
</tr>
<tr>
<td>buildResp</td>
<td>Prepares the response by adding an appropriate status code, a list of URLs and HTTP headers for caching.</td>
</tr>
<tr>
<td>rmvExpr</td>
<td>Removes entries that have expired. Ports in more than one component.</td>
</tr>
<tr>
<td>sndQryG</td>
<td>Sends query to cache to retrieve an entry and its version number.</td>
</tr>
<tr>
<td>sndQryU</td>
<td>Sends query to cache to update an entry and its version number.</td>
</tr>
</tbody>
</table>

Fig. 7: Semantics of key transitions in $A_{REST}$

VI. COMPOSITE WEB SERVICE ARCHITECTURE

In this section, we focus on the problem of composing an architecture for a SOAP-based Web Service with an architecture for replicating the service’s session state, so that the two architectures do not interfere with each other.

A. Superposition of non-interfering architectures

When a Web Service is designed based on a formal architecture, we enforce the characteristic properties of this architecture. Two architectures should be possible to be composed, so that their properties are preserved in the composite architecture. Such a design approach would realize the principle of separation of concerns. We currently lack a formally defined composition operator for guaranteeing deadlock-freedom and non-interference of the composed architectures.

For this reason, we discuss sufficient conditions for the correct-by-construction superposition of two architectures, whose behavior encompasses all control locations needed for their coordination. Let us consider that the superposition of architectures $A[C_1, \ldots, C_n]$ and $B[C_1, \ldots, C_n]$ results in a new architecture $(A + B)[C_1, \ldots, C_n]$ defined over the same set of global states. Any two interactions $a$ and $b$ from $A$ and $B$ respectively, which may be concurrently enabled at some state $q$, do not have to interfere in $A + B$. This is guaranteed for any of the following two conditions:

1) whenever $a$ is enabled, $b$ is enabled too that is: $\exists q'. q \xrightarrow{a} q' \iff \exists q''. q \xrightarrow{b} q''$

2) we have commutativity: $q \xrightarrow{a} q' \xrightarrow{b} q_f$ and $q \xrightarrow{b} q'' \xrightarrow{a} q_f$

These two conditions guarantee that the result of synchronizing the interactions $a$ and $b$ will not introduce any deadlock. In the architecture $A + B$, we will have $q \xrightarrow{ab} q_f$.

B. Web Services with Session Replication

Services may implement stateless interactions with their clients or they can keep a session state for each of them. In the latter case, clients are assigned a session identification and the server maintains, for each session, information spanning over multiple task executions. The goal of session replication is to replicate the session state of services in a cluster of replicas (i.e. servers), thus offering transparency at fail-over.

The main variation points of an architecture for session replication are the means used for in-cluster communication and session state storage. For example, architecture variants may involve multicast in-cluster communication, memory to memory session replication etc.

We define an architecture $A_{SREP}$ for a session replication with sticky sessions (i.e. where all requests of one client are routed to the same replica) in a cluster where replicas send periodic heartbeats to a registry and the session state is stored in a shared database.

$A_{SREP}$ aims to enforce the characteristic property:

$$P_{SREP} = P_1 \land P_2 \land P_3 \land P_4$$

where

- $P_1$: services assign and maintain sticky sessions
- $P_2$: available servers can be discovered
- $P_3$: server fail-overs are transparent to clients
- $P_4$: session state can be transferred among servers

The model for $A_{SREP}$ shown in Fig. 8 includes a proxy service ($srvc_{SC}$) that routes requests to the $k$ replica
services \((\text{srvc}_{-}^{s_{C_{r}}})\). The \text{srvc}_{-}^{c_{C_{p}}} includes a registry \((\text{reg})\), multiple proxy tasks \((\text{task}_{-}^{c_{C_{p}}})\) and a data component \((\text{data}_{C_{p}})\) which stores the session state in a transactional manner. The \text{task}_{-}^{c_{C_{p}}} implements the logic for routing an invocation, while the \text{reg} is responsible for detecting and broadcasting replica failures. Every \text{srvc}_{-}^{s_{C_{r}}} includes a replica listener \((\text{listn}_{C_{r}})\) that sends heartbeats and multiple replica tasks \((\text{task}_{-}^{s_{C_{r}}})\) with instructions for reading/writing the session state.

For \(k\) replicas \((\text{srvc}_{-}^{c_{C_{r}}})\) \(A_{SREP}\) is defined as follows:

\[A_{SREP}(k)\]
\[
\text{srvc}_{-}^{c_{C_{p}}}\{\text{invoc}_{-}^{endpt} + \text{task}_{-}^{c_{data}}\}
\text{srvc}_{-}^{c_{C_{p}}}\{\text{invoc}_{-}^{endpt} + \text{dispcht} + \text{task}_{-}^{c_{C_{p}}} + \text{data}_{C_{p}}\ \text{reg}\},
\text{srvc}_{-}^{s_{1}}\{\text{task}_{-}^{s_{C_{r}}} + \text{listn}_{C_{r}}\ \text{data}_{C_{p}}\ \text{dispcht}+\}
\ldots \text{srvc}_{-}^{s_{k}}\{\text{C}_{C_{r}},
\text{network}\}
\]

The glue in Fig. 8 (ports are defined in Fig. 9) applies a set of architecture constraints to enforce the characteristic property of \(A_{SREP}\). More precisely, the constraints that enforce \(P_{1}\) are: (1) \text{task}_{-}^{s_{C_{r}}} stores session ids for new requests to \text{reg} and returns them to clients. (2) \text{task}_{-}^{s_{C_{r}}} stores every session’s state in \text{data}_{C_{p}} and resets the sessions’ idle time upon receiving their requests. (3) \text{reg} maintains and gives access to a list of alive replicas and their idle time (i.e. elapsed time since last request). (4) \text{task}_{-}^{c_{C_{p}}} associates a session with an alive replica and forwards all the session’s requests to this replica. (5) \text{reg} periodically removes sessions that have expired (i.e. whose idle time has exceeded the maximum allowed idle time) also deleting them from \text{data}_{C_{p}}.

Property \(P_{2}\) is enforced by the constraints: (1) \text{reg} maintains and provides access to a list of alive replicas and their idle times. (2) Periodically, \text{reg} detects replica failures by checking the idle times and broadcasts these failures to all \text{task}_{-}^{c_{C_{p}}}. (3) All \text{listn}_{C_{r}} send signals of aliveness to \text{reg}, which resets their idle time.

The constraints for \(P_{3}\) are: (1) \text{task}_{-}^{c_{C_{p}}} receives client requests, forwards them to a \text{srvc}_{-}^{s_{C_{r}}} and sends the responses back to clients. When it expects a response from a failed \text{srvc}_{-}^{s_{C_{r}}}, it sends a failure notification, it forwards the request to another \text{srvc}_{-}^{s_{C_{r}}}. (2) The multiple \text{task}_{-}^{s_{C_{r}}} store request results (with checkpoints) in \text{data}_{C_{p}} and do not perform their function (and effects on session state) for requests with stored results.

The constraints for \(P_{4}\) are as follows: (1) \text{task}_{-}^{s_{C_{r}}} searches every request’s session state first in \text{data} (local store) and then in \text{data}_{C_{p}} (global store). (2) Before returning the response, \text{task}_{-}^{s_{C_{r}}} stores atomically (transactional mode) in \text{data}_{C_{p}} the altered session state and the checkpoint.

C. Correctness and non-interference in \(A_{SREP} + A_{SOAP}\)

By establishing the correctness of \(A_{SREP}\) with respect to its characteristic property we exclude problems like the loss of session state. Such a functional safety property can be easily checked by state exploration of the BIP model, as in [7].

The superposition of \(A_{SOAP}\) and \(A_{SREP}\) denoted by \(A_{SREP} + A_{SOAP}\) merges \(g_{SOAP}\) and \(g_{SREP}\). Any two
interactions $gS$ of $gS_{SOA}$ and $aC$ of $gS_{REP}$ that may be concurrently enabled at a global state $q$ of $A_{SOAP} + A_{REP}$ do not interfere. This is true because:

- All interactions $gS$ and $aC$ that remain independent in $A_{REP} + A_{SOAP}$ are commutative. In fact, the only case when this may not hold is if a single component enables concurrently some $gS$ and $aC$. However, interactions $gS$ are enabled exclusively by the $invC_{endpt}$ and $dispC_{h}$ components, which do not enable any $aC$.
- Any $gS$ and $aC$ that are synchronized in $A_{REP} + A_{SOAP}$ are always enabled together and are commutative.

**VII. Case Studies**

We present two cases of correct-by-construction Web Services and their executable BIP models built by applying the design steps of section V-A. The two services are based respectively on $A_{SOAP}$ and $A_{REST}$ and have been checked for deadlock-freedom using the BIP state exploration tool. The BIP models and the Web Services can be accessed in http://depend.csd.auth.gr/ServiceSystemsModelling.php.

**A. A Web Service based on $A_{SOAP}$**

Our SOAP-based Web Service for online management of shopping baskets (carts) is represented by a $srvc_s$ BIP component that includes three $task_s$: (i) $task_s_{Crt}$, which creates a cart (ii) $task_s_{Add}$, which adds an item to a cart and (iii) $task_s_{Clr}$, which clears a cart.

**Fig. 10:** Input and output of the implemented $task_s$

In Fig. 10, the input and output of the $task_s$ components are shown. The $task_s_{Crt}$ must be invoked first, in order the client to obtain the session id of the subsequent communication. Upon invocation, the $task_s_{Crt}$ creates a new cartId, which is returned to the client. Then, the $task_s_{Add}$ may be invoked, which adds the item $itemld$ to the cart cartId. If cartId indicates an expired session, an error and a new cartId are returned to the client. The $task_s_{Clr}$ removes all items from a cart.

The session state for each client is maintained in a $data_s$ component. The $data_s$ stores two table structures, one for the cartIds and their idle times and one for the associated itemldS and cartIds.

The BIP model also includes a $srvc_c$ with a $task_c$, that invokes service tasks according to the expected invocation sequence, i.e. $task_s_{Crt}, task_s_{Add}, task_s_{Clr}$. A second $srvc_c$ includes a $task_c$ that invokes $task_s_{Clr}, task_s_{Add}, task_s_{Crt}, task_s_{Add}, task_s_{Add}$ in this order. The model has been deployed as a Web Service under the control of the BIP execution engine that invokes message processing functions generated by the gSOAP tooset [30].

**B. A Web Service based on $A_{REST}$**

Our model for a RESTful Web Service consists of a $srvc_s$ that includes two $task_s$: (i) the $task_s_{get}$ for viewing a list of available items and (ii) the $task_s_{post}$ for adding an available item to an existing cart, while making the item unavailable for other clients.

Requests are issued in parallel by $task_c_{R1}$ and $task_c_{R2}$, which are included in two separate $srvc_c$. Initially, $task_c_{I}$ issues a request for $task_s_{get}$ and stores the result to local cache. Subsequently, it invokes $task_s_{post}$. On the other hand, $task_c_{I}$ issues two sequential requests for $task_s_{get}$.

The $task_c_{1}$ and $task_c_{2}$ components realize scenarios where (i) the client does not own a cached result (ii) the client owns an old cached result and (iii) the client owns a fresh cached result. The composite model with the aforementioned components was also found deadlock-free.
VIII. CONCLUSION

We presented an approach for the design of correct-by-construction Web Services. Our work, inspired by the recent developments in the BIP component framework, applies the principles of property enforcement and property composability in the design of Web Service architectures. In section V-A we discussed the steps of a design flow based on the developed BIP models for SOAP-based and RESTful Web Service architectures. Regarding the challenge of composing architectures that enforce specific properties, we examined the superposition of one architecture over another one and the associated problem of non-interference between their interactions. In this way, we were able to derive a SOAP-based Web Service architecture with session replication. As a proof of concept, we discussed the development of one SOAP-based Web Service and one RESTful service based on the design flow of section V-A.

The overall contribution constitutes a new proposal in the design of Web Services, as opposed with the current design practice that was outlined in section III.

In our future plans we aim to combine the new design flow with the result of our work in [7] towards the development of correct-by-construction service compositions. We also aim to utilize the BIP extensions for the modeling and analysis of Web Service architectures with dynamic reconfigurations [16].

ACKNOWLEDGMENTS

We thank Prof. J. Sifakis and the BIP group in VERIMAG-Grenoble for having introduced us to the BIP technical approach and for their support. This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program “Education and Lifelong Learning” of the National Strategic Reference Framework (NSRF) - Research Funding Program: Thalis Athens University of Economics and Business - SOFTWARE ENGINEERING RESEARCH PLATFORM.

REFERENCES