Composite Web Services as Dataflow Graphs for Constraint Verification

Jyotsana Gupta, Joey Paquet and Serguei A. Mokhov
Concordia University, Montreal, Quebec, Canada

Keywords: Composite Service, Service Constraint Verification, Service Context.

Abstract: Owing to advantages such as re-usability of components, broader options for composition requesters and liberty to specialize for component providers, composite services have been extensively researched and significantly enhanced in several respects. Yet, most of the studies undertaken fail to acknowledge that every web service has a limited context in which it can successfully perform its tasks. When used as part of a composition, the restricted context-spaces of all such component services together define the contextual boundaries of the composite service. However, from a thorough review of the existing literature on the subject, we have discovered that no systems have yet been proposed to cater to the specific verification of internal constraints imposed on components of a composite service. In an attempt to address this gap in service composition research, we propose a multi-faceted solution capable, firstly, of automatically constructing context-aware composite web services with their internal constraints positioned for optimum resource-utilization and, secondly, of validating the generated compositions using the General Intensional Programming SYstem (GIPSY) as a time- and cost-efficient simulation/execution environment.

1 INTRODUCTION

Web services are independent, self-describing, modular programs that can be published, searched for, invoked, and executed via the World Wide Web. In order to enhance their clarity and re-usability, web services are usually designed to perform simple and specific tasks. To accomplish more complex tasks, such as a shopping process comprising of product catalog display, customer order gathering, credit card payment processing and product shipment initiation (as depicted in Figure 1), simple web services (called *component services*) are selected for each sub-task and composed together in the form of a workflow to build a *composite web service*, which is a customized service assembled and arranged according to each client's particular and potentially elaborate requirements. The process of *web service composition*, besides broadening the client base for each service provider, opens up a wide variety of service options to choose from for each sub-task for the service users, thereby benefiting both of the parties involved in the transaction.

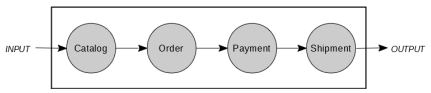

Figure 1: Composite Online Shopping Service.

Owing to such advantages, among others, service composition has been extensively researched. Several aspects of the composition process have been exhaustively explored and honed. Yet, there are certain facets of composite web services that have not been granted proper consideration. For example, almost all of the existing research on web service composition overlooks the fact that every service has a limited context (albeit possibly wide) in which it can successfully perform its tasks. Such limitations are defined by service providers and are called *internal service constraints*. For example, while one credit card payment service (say, W_1) might be capable of processing only Visa cards (internal constraint: $CreditCardBrand = Visa$), another service (say, W_2) might be able to process both Visa and Master cards (internal constraint: $CreditCardBrand \in \{Visa, Master\}$). Similarly, a shipment service (say, W_3) might be capable of delivering products only to addresses within Canada (internal constraint: $ShippingAddress \in \{Canada\}$). It cannot be assumed that services can process every existing credit card brand or deliver products to every country in the world. Furthermore, if the services W_1 and W_3 mentioned above were to be used as components of the composite online shopping service depicted in Figure 1, the shopping service as a unit would be constrained to $\{CreditCardBrand = Visa \cap ShippingAddress \in \{Canada\}\}$, i.e., it would

Gupta, J., Paquet, J. and Mokhov, S.
Composite Web Services as Dataflow Graphs for Constraint Verification.
DOI: 10.5220/0009342302820291
In *Proceedings of the 10th International Conference on Cloud Computing and Services Science (CLOSER 2020)*, pages 282-291
ISBN: 978-989-758-424-4

be useful for only those customers who reside in Canada and use a Visa card for online transactions. Clearly, this limits the customer-base of the online store employing the composite shopping service. Due to such impact, it is important to take all the internal constraints of atomic services into account while studying the behavior of composite services. Internal constraints in the context of service composition have been discussed in elaborate detail only by Wang, Ding, Jiang, and Zhou in (Wang et al., 2014), followed by Laleh, Paquet, Mokhov, and Yan in (Laleh, 2018; Laleh et al., 2018; Laleh et al., 2017).

Due to the limited exposure received by this aspect of web service composition, no systems (as per our knowledge) have yet been proposed to cater to the specific verification and validation of constraints imposed on component atomic services by their developers/providers. Most of the existing research on verification/simulation/execution of composite web services has been concerned with validating their Quality of Service (QoS) constraints (Shkarupylo, 2016), functional requirements (Dechsupa et al., 2016) or order of execution of the component services (Huynh et al., 2015).

The internal service constraints that we aim to verify are different from those discussed in the research works cited above. These are restrictions imposed on the context in which a service can be executed. We borrow the concept of *execution context* of a web service from (Laleh, 2018) and consider it to be an aggregation of all the information that could affect the execution of a service. According to this definition, each element of an execution context is a name-value pair. Based on that, the context of a service (be it atomic or composite) is considered to be the set of all its input parameters and the values that are assigned to them at the time of a service call. For example, consider the composite shopping service depicted in Figure 1 and its component details for a sample run provided in Table 1. For this sample run, the context of atomic service W_2 would be:

$$\{ProductNumber : ST1234,\ ProductPrice : 75.00\}$$

while the context of composite shopping service (assuming that it takes *ProductName, CreditCardBrand, CreditCardNumber and ShippingAddress* as input) would be:

$$\{ProductName : StudyTable,\ CreditCardBrand : Visa,\ CreditCardNumber : CCVS56789,\ ShippingAddress : Canada\}$$

For any composite service, many of the context parameters/dimensions of its component services could get their values assigned dynamically as the composite service is being executed. Therefore, in order to check if the restrictions placed on such variables (i.e., the internal constraints) are satisfied, we must either actually execute the composite service or else simulate its execution. For example, suppose, the Shipment service provider imposes a 50-pound weight-limit on packages shipped by the service (internal constraint: $ProductWeight <= 50$). In that case, the *ProductWeight* output parameter produced by the Catalog service (which is an input to the Shipment service) would have to be inspected for values exceeding 50 before the Shipment service could confirm acceptance of the shipment job. To accomplish that, the composite service would have to be executed, which would produce some value for the *ProductWeight* parameter to be compared with the shipment weight-limit. Such verification demands a simulation or execution environment capable of executing composite web services composed of either simulated or real-world, internally-constrained atomic web services.

In an attempt to address the above concerns related to web service composition, in this paper, we propose the use of the General Intensional Programming System (GIPSY) (Paquet, 2009) as a simulation/execution-based environment for verification and validation of constraint- and context-aware composite web services.

1.1 Objectives and Motivation

Consider an online store that needs a shopping service that can display a product catalog to its customers, accept their orders, process their credit card payments and initiate shipment of the ordered goods. Such a store effectively needs a composition engine that can construct the required shopping service (similar to the one depicted in Figure 1) based on the inputs that the store's customers would be able to provide, the outputs that they would expect in return and the set of atomic services available to perform each of the required tasks. However, as discussed earlier, atomic services may have restrictions, known as internal constraints, imposed on their execution context by their providers, which, in turn, defines the contextual boundaries of any composite service of which they form a part. Table 1 specifies such internal constraints placed on the Payment and Shipment services that serve as components of the shopping service. The combinatorial effect of these constraints transforms the shopping service into a utility tailor-made for customers who use Visa credit cards for online transactions, order products weighing up to 50 lbs and get their purchased goods delivered within Canada, which makes it essential for the composition engine

Table 1: Online Shopping Service Component Details.

Service	Type	Input Parameters	Sample Input Values	Output Parameters	Sample Output Values	Internal Constraints
W_1	Catalog	{ProductName}	{StudyTable}	{ProductNumber, ProductPrice, ProductWeight}	{ST1234, 75.00, 45}	$C_1 = \emptyset$
W_2	Order	{ProductNumber, ProductPrice}	{ST1234, 75.00}	{OrderNumber, PaymentAmount}	{ORD1234, 82.50}	$C_2 = \emptyset$
W_3	Payment	{OrderNumber, PaymentAmount, CreditCardBrand CreditCardNumber}	{ORD1234, 82.50, Visa, CCVS56789}	{PaymentStatus}	{Complete}	$C_3 = \{CreditCardBrand = Visa\}$
W_4	Shipment	{PaymentStatus, ProductWeight, ShippingAddress}	{Complete, 45, Canada}	{ShipmentStatus}	{Confirmed}	$C_{41} = \{ProductWeight <= 50\}$ $C_{42} = \{ShippingAddress = Canada\}$

to take the internal constraints of the Payment and Shipment services into account while assembling the shopping service.

Continuing with the motivation scenario, once the composition engine assembles a suitable composite shopping service, it must subject the service to some basic behavioral tests, such as validation of input-output relationships, correct constraint enforcement, and elimination of inaccessible or malfunctioning components, before it can be proposed as a practicable solution to the service requester, i.e., the online store. However, as mentioned in Section 1, while there are systems available for testing composite service behaviors in an unlimited context space, no existing research offers a solution for verification and validation of internally-constrained composite web services. Therefore, the composition engine in this scenario has no option other than to trust the component services to function well within their bounded context space (as advertised in their descriptions) and to present the composed shopping service to the service requester without testing its contextual limits. Similarly, the service requester has no alternative but to trust the composition engine to have performed all possible checks and to accept the composed solution as a feasible one, following which it might enter into a binding contract or lease with the providers of each of the component services involved. Let us suppose that the Payment component service, W_3, has a faulty implementation or its constraint description was mistakenly replaced by that of a different Payment service. Because of one or more of such human-errors, W_3 ends up being presented as a Visa card processing service while actually having been programmed to process Master cards. Now, suppose that the online shopping store is not allowed to accept Master cards because of an agreement with Visa. In this situation, if a customer of the store attempts to make an online purchase with their Visa credit card, the Payment service would fail and the order would be rejected. Meanwhile, another customer using a Master card might succeed in placing their order, thus violating the store's agreement with Visa. Not only would such an event prevent the store from making any valid sale until the service is repaired or replaced, it could have an adverse effect on the store's reputation, which could harm its business. Additionally, the store might have to suffer monetary losses because of the contract signed with the provider of the Payment service (W_3), as it no longer holds any utility for it, or even face legal consequences for their breach of agreement with Visa. Besides, the composition engine is also likely to get affected by such occurrences and lose its credibility with its clients for supplying unreliable and defective composite services.

As explained earlier, contextual elements of component services often get their values assigned dynamically as their composite service is being executed, which implies that any conditions placed on those elements must be evaluated at run-time. Therefore, a system meant for the verification of context- and constraint-aware composite web services must be capable of either simulating execution of or actually executing those composite services.

While with the execution system we aim to test the actual behavior of services and weed out non-functional components or components that do not behave according to their agreed constraints/specifications, our purpose for the simulation system is to be able to test the suitability of composition solutions and implement quick fixes in case of issues with minimal resources and time and without requiring access to any service's code.

We recognize an enhancement in time-efficiency by a reduction in the average response-time of a composite service both during simulation and execution. An increase in cost-efficiency, on the other hand, can have two aspects: a reduction in the average fee to be paid to the component service providers for utilizing (i.e., executing) their services as part of a composition or a decrease in the number of component service rollbacks to be borne due to constraint-violation

during composite service execution.

1.2 Literature Review

The primary goal of this paper is to present a simulation/execution-based verification solution for context- and internal-constraint-aware composite services. We present here some related solutions.

The research conducted by Wang et al. (Wang et al., 2014) and Laleh et al. (Laleh, 2018; Laleh et al., 2018; Laleh et al., 2017) on composition of internally-constrained web services. These researchers propose graph-search-based composition algorithms augmented with novel techniques to merge composition plans/sub-plans operating in different contexts into a single larger plan for widening the overall contextual range of a composite service. However, neither of these solutions can guarantee exhaustive coverage of a context space, which leaves the generated composite services vulnerable to run-time constraint-verification failures, thereby necessitating the construction of a behavior verification and validation system. Using Petri nets for composition of internally-constrained services allows for such a verification to a certain extent by means of simulation. For example, Cheng, Liu, Zhou, Zeng, and Yla-Jaaski (Cheng et al., 2015) introduce an automated composition method for internally-constrained fuzzy semantic web services using Fuzzy Predicate Petri Nets. Such approaches ensure that the generated composite services satisfy the input/output requirements of the user while ensuring that they do not conflict with the internal constraints of component services. However, Petri nets are not capable of executing real services and, therefore, cannot be used for execution-based verification of composite services.

Most of the simulation/execution solutions use WS-BPEL representation of composite services as their input, which they first translate into a verification language or a formal simulation model and then validate them using a simulator or model checker tool. For example, Siala, Ait-Sadoune, and Ghedira (Siala et al., 2014) propose translation of WS-BPEL processes into Multi-Agent Systems, which can simulate service behavior using JADE framework for observation and detection of undesired properties, such as live-lock and deadlock. Meanwhile, Juan and Hao (Juan and Hao, 2012) use QPME tool for, first, transforming WS-BPEL processes into Queuing Petri Nets (QPN's) and, then, simulating the QPN's for quantitative performance analysis. Dechsupa, Vatanawood, and Thongtak (Dechsupa et al., 2016) suggest Colored Petri Nets for formal modeling of WS-BPEL processes and CPN Tool for editing, simulation and analysis of the CPN's generated. Translation of composite services written using WS-BPEL into Promela language is discussed by Nagamouttou et al. (Nagamouttou et al., 2015). SPIN tool is used for analyzing these translations for various desirable and undesirable properties. Chen, Tan, Sun, Liu, and Dong (Chen et al., 2014) propose using VeriWS for analyzing the semantics of a WS-BPEL process directly (i.e., without translation) to check for deadlocks, reachability, and QoS constraint-satisfaction as well as for detecting anomalies in the composition's functionality. Shkarupylo (Shkarupylo, 2016) proposes synthesis of formal TLA+ specification for a WS-BPEL composition and testing its functional and non-functional properties using TLC model checker and DEVS Suite toolkit through simulation and visualization of service behavior. Although several of these approaches succeed in incorporating automation and extensibility into their translation processes (similar to ours), none of them supports completely automated composition of services, which limits their practicality while handling complex compositions or large repositories of services.

2 SERVICE COMPOSITION

While simulating and executing internally-constrained composite services is the primary goal of this paper, constructing such services based on a composition request and a set of services available for composition is an essential prerequisite to the simulation/execution process. The automated service composition technique that we employ in this paper has been borrowed from the research conducted by Laleh et al. (Laleh, 2018; Laleh et al., 2018; Laleh et al., 2017; T. Laleh et al., 2017) for its formal model of constraint-aware composite services and its unique constraint-adjustment technique, which enhances the time- and cost-efficiency of the composition and, subsequently, the simulation/execution process.

To aid with a better understanding of the service composition methodology devised by Laleh et al., in this section, we present the formal definitions of the fundamental entities and concepts involved in the process (Laleh, 2018).

Definition 1. *A **Service** is a tuple* $S = \langle I, O, C, E, QoS \rangle$ *where:* $\boldsymbol{I}$ *is the set of ontology types representing the input parameters of the service;* $\boldsymbol{O}$ *is the set of ontology types representing the output parameters of the service;* $\boldsymbol{C}$ *is the set of constraint expressions representing limitations on service features;* $\boldsymbol{E}$ *is the set of ontology types representing parameters whose values are affected as*

a result of the execution of the service; ***QoS*** *is the set of quality parameters of the service.*

Definition 2. *An* ***Internal Constraint*** *is a boolean expression of the form:* ⟨**feature**⟩ ⟨**operator**⟩ ⟨**literalValue**⟩*, where:* ⟨**feature**⟩ *represents an input parameter of a service, which is an ontology type;* ⟨**operator**⟩ *represents operators such as* $=, <, >, \leq, \geq$*;* ⟨**literalValue**⟩ *represents a value or a set of values of the same data type as the expression feature.*

Definition 3. *A* ***Service Composition Request*** *is a tuple* $R = \langle I, O, QoS, C \rangle$ *where:* ***I*** *is the set of ontology types representing the input the customer can provide;* ***O*** *is the set of ontology types representing the output expected by the customer;* ***QoS*** *is the set of quality parameters expected from the service by the customer;* ***C*** *is the set of constraints representing limitations of service requester.*

Since, in this paper, we take only internal constraints into consideration, our current implementation of a composition request models but does not process the requester's QoS and constraint requirements. In response to a composition request, we generate a set of one or more *solution plans*, i.e., workflows of component services capable of producing the requested output by processing the given input while verifying the internal constraints placed on the components. These plans are called constraint-aware plans.

Definition 4. *A* ***Constraint-Aware Plan*** *is a directed graph extracted from the search graph in which each node is a service-node* $\langle C_S, service \rangle$*, using initial parameters (R.I), whose successive application of services of nodes eventually generates the goal parameters (R.O).*

The search graph referred to in the above definition is the graph of service nodes that gets generated as a result of the forward expansion stage of the composition process and represents a collection of all the solution plans that can be constructed for a given composition request. For each service-node $\langle C_S, service \rangle$ in a constraint-aware plan, C_S refers to the set of all service constraints that must be verified before *service* can be executed as part of the plan. The term *R.I* refers to the input parameters specified as part of the given composition request *R* while *R.O* refers to the requested output parameters. Each service-node in a constraint-aware plan has a set of predecessors and a set of successors associated with it, which are defined as follows:

Definition 5. *The* ***predecessor*** *set of a service-node in a constraint-aware plan represents the set of all services-nodes that must be executed before the execution of the service-node, and the* ***successor*** *set represents the set of all services-nodes that will be executed only after the execution of the service-node in the constraint-aware plan.*

We present our service composition methodology as a set of algorithms, which transform a given composition request and set of available services into a set of constraint-aware composition plans or composite services.

The main **ServiceComposition** algorithm drives the service composition process: (1) *forward expansion* constructs a *search graph* based on a given composition request and available services; (2) *backward search* extracts *solution plan sets* from the search graph; (3) *plan construction* discards extraneous services from solution plan sets and arranging the remaining ones into *solution plans*, and (4) *constraint-aware plan construction* transforms solution plans into *constraint-aware plans* with their constraint verification points adjusted to optimum locations.

The **ForwardExpansion** algorithm generates a search graph for a given composition request. A search graph is a directed graph composed of ordered layers, each of which is assigned certain specific services selected from the given repository. As forward expansion begins, *prdSet* is initialized with the initial parameters, after which the repository is searched for those services all of whose input parameters exist in *prdSet*. Each of the suitable services discovered during the search must generate an output parameter that is not already present in *prdSet* and must not violate any of the requested constraints. Services that match these criteria are added to the next layer in the search graph. The output parameters produced by all the services included in the layer are then added to *prdSet* and the repository is searched again for services that can be added to the following layer based on the updated set of parameters. The search graph thus grows layer by layer until no more services from the repository can be added. If the *prdSet* obtained at the end of the expansion contains all the goal parameters (*R.O*), the problem is solvable and the search graph is returned to the ServiceComposition algorithm for further processing.

The **BackwardSearch** algorithm recursively extracts service sets that can reach the goal parameters from the initial parameters from a search graph using a backward-chaining strategy. The algorithm begins by generating all possible combinations of the services in the last layer of the search graph. For each of these combinations, it extracts their predecessors from the previous layer and generates all possible combinations. The process is repeated for these and all subsequent combinations until the first layer is reached. Thus, a large number of branches or se-

quence of sets of services (*planSet*) going from the first to the last layer of the search graph are created. For each of these sets, the set of outputs of all its services is checked to ensure that it includes all the goal parameters. Each *planSet* that is successfully verified is returned to the Service Composition algorithm for further processing while others are discarded as invalid. This process is repeated with each layer of the search graph serving as the starting layer for backward search.

The **ConstructPlans** algorithm first organizes the services in a *planSet* from the BackwardSearch algorithm in order of their layer indexes to create a layered directed graph, or plan. Then, for each service in the plan, the set of outputs of all its predecessors present in the plan together with the initial parameters is checked to ensure that it includes all the inputs of the service, otherwise, the service is removed from the plan. Each of the remaining services in the plan that neither have a successor in the plan nor produce any goal parameter are also discarded. This removal process continues until no more services can be removed from *plan*. If the set of outputs of all remaining services in the plan includes all the goal parameters, the plan is returned to the ServiceComposition algorithm for further processing.

The **ConstructCAPlans** algorithm first transforms the solution plans generated by ConstructPlans into *constraint-aware plans* and then adjusts each constraint of each service-node (*serviceNode*) in each plan to an optimal location. To accomplish that, a predecessor service-node (*preNode*) to *serviceNode* that affects the value of the feature to which the constraint applies is selected. The constraint is then moved to verification points immediately before the execution of all successor service-nodes of *preNode*. This process is repeated with all the predecessors of *serviceNode* as well as their predecessors until the constraint is moved to the earliest and most efficient verification point in *cnstrAwrPlan*. In case no predecessors are found to affect the constrained feature's value, the constraint is moved to the beginning of *cnstrAwrPlan*. Once all the constraints in all *cnstrAwrPlans* are adjusted, they are returned to the main ServiceComposition algorithm as constraint-aware solutions to the given composition request.

3 TRANSLATION TO OBJECTIVE LUCID

We propose the use of GIPSY as the simulation/execution environment for context- and constraint-aware composite services. However, since GIPSY is dedicated to the compilation and execution of LUCID programs (Ashcroft et al., 1995; Wadge and Ashcroft, 1985), composite services, before being executed on GIPSY, must be translated into a LUCID dialect, which, in our case, is OBJECTIVE LUCID (Mokhov, 2005; Mokhov and Paquet, 2005). To accomplish that, we design and implement a flexible translator framework capable of allowing modular plugging-in and -out of different translator programs, as and when required. This framework can accept a composite service as input through an extensible multi-modal input system and translate it into any of the target models for which a translator module is available. There are several reasons responsible for our decision to use OBJECTIVE LUCID for representing and GIPSY for simulating/executing context- and constraint-aware composite services, which we discuss below:

LUCID, being an intensional programming language, it can efficiently represent context-dependent entities, allowing each context to be composed of infinite-many dimensions and each dimension to be defined as a regular variable in a concise and precise manner besides providing operators # and @ for directly extracting values from and specifying values for the contextual dimensions respectively. Additionally, the *whenever* operator supplied by LUCID enables context-dependent conditions (i.e., internal constraints) to be placed on variable and function definitions (i.e., services), allowing them to be computed/executed only if the conditions evaluate to *true* (Ashcroft et al., 1995; Wadge and Ashcroft, 1985).

OBJECTIVE LUCID is a hybrid language that employs LUCID for representing a composite service as a dataflow network and JAVA for specifying the operations performed by component services, thereby combining the ease and enhanced productivity of programming in mainstream languages with the efficiency of intensional dataflow languages (Mokhov, 2005; Mokhov and Paquet, 2005). In addition to that, its use of JAVA enables a simulated function definition for a component service to be easily swapped with a definition that invokes the real online service, thereby allowing the verification process to be conducted through simulation as well as execution while minimizing the effort involved in switching between the two modes.

Since LUCID is a dataflow programming language, upon execution, a LUCID program representing a constraint-aware composite service is interpreted as a dataflow network of concurrently-executing component service filters enveloped in wrappers serving as internal-constraint-verification layers. GIPSY's eductive, demand-driven approach

towards execution together with its warehouse unit capable of storing and being queried for execution results computed earlier in specific contexts (instead of re-evaluating them) significantly reduce the overall time, effort and cost spent on simulation/execution of composite services (Paquet, 2009; Ashcroft et al., 1995).

The implementation and operation of our OBJECTIVE LUCID translator is based on a set of algorithms that together define our translation methodology. In this section, we present and explain these algorithms using the shopping composite service depicted in Figure 2.

The **TransCSToOLucid** algorithm drives the translation process, invoking the other algorithms when required. It consists of four major steps: (1) *ValidateInpValues* responsible for ensuring that each composite service input parameter receives a value of appropriate data type, (2) *GenerateJavaSegment* for producing the JAVA segment of the translation, (3) *GenerateOLucidSegment* tasked with generating the LUCID segment of the resultant program, and (4) appending the generated JAVA segment with the LUCID segment for constituting the complete OBJECTIVE LUCID translation of the given composite service.

The **GenJavaSegment** algorithm is responsible for generating the JAVA segment of the OBJECTIVE LUCID translation of a composite service. This segment, comprises of a collection of JAVA class and method definitions representing two types of nodes: the output accumulator node and the component service nodes. The output accumulator node represented as JAVA class *CAWSReqComp*, where *CAWS* stands for *Constraint-Aware Web Service*) assembles composite service outputs generated by one or more component services as data members of a single object, which can be returned as the computed value of the composite service's LUCID expression, thereby allowing the composite service to produce multiple outputs in any given context.

The **GenAtomSvcJavaDef** algorithm is invoked by GenJavaSegment for generating a pair of JAVA class and free function definitions that together define and provide the means of triggering the operation of a component service. For example, consider the call to service *W4*'s free function *w4*. User input *shippingAddress* and outputs *productWeight* and *paymentStatus* produced by components *W1* and *W3* respectively, which serve as inputs to *W4*, are supplied as arguments to this function call to be used by the function as arguments while calling the corresponding service class (*CAWSW4*) constructor. The constructor initializes *W4*'s input parameters acting as *CAWSW4*'s data members with these arguments while initializing the output data members with dummy values. Once service object *oCAWSW4* is created, method *w4* uses it to invoke *CAWSW4*'s *process* member method responsible for processing the given inputs and updating the output data members with the results obtained, following which method *w4* returns the updated *oCAWSW4* object and, hence, the service outputs to the LUCID segment. The algorithm begins by defining the given component service's JAVA class, comprising of its input and output parameters as data members, a constructor to initialize them, and *process* member method to transform the inputs into outputs. Then the algorithm generates the component service's free function definition.

The **GenOLucidSegment** algorithm builds the LUCID segment of the OBJECTIVE LUCID translation of a composite service. The algorithm starts by defining the main LUCID expression, representing the outcome of the composite service, along with the global execution context, comprising of composite service input name-value pairs leading to context-driven execution. Then the algorithm constructs the LUCID expression representing the output accumulator node of the composition. This node accepts composite service outputs as inputs or arguments to its JAVA function call, which assembles them into an object. Values for the arguments are extracted from the local context of the accumulator's LUCID expression defined as a set of composite service output parameter names paired with their respective values generated by various component services. The LUCID representation of each of the component services, which also forms part of the accumulator node's *where* clause, is generated by the GenAtomSvcLucidDef algorithm invoked iteratively from the GenOLucidSegment for each service-node in the composition's constraint-aware plan.

The **GenAtomSvcLucidDef** algorithm is invoked by GenOLucidSegment for generating the LUCID representation of a component service. It begins by defining the set of the given service-node's input parameters and constraint-features as the contextual dimensions for its LUCID expression. Since, before computing a component service expression in LUCID, all the constraints attached to its service-node must evaluate to *true*, constraint-features must be included in the component service's dimension list as well as evaluation context, which specifies the source of their values – essential for their computation. The algorithm builds this evaluation context, pairing the component's dimensions with values either received from the user (extracted from global dimensions using # operator) or generated as outputs by other compo-

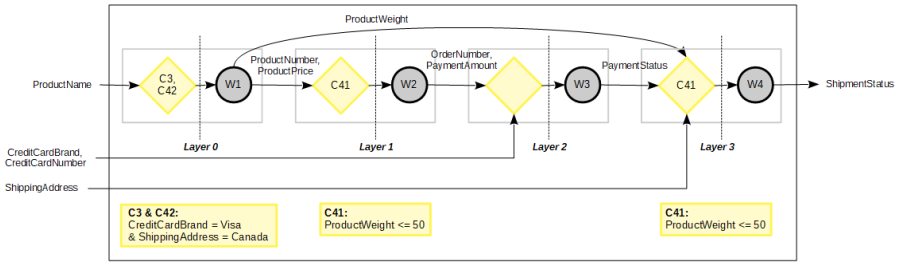

Figure 2: Constraint-aware Shopping Composite Service.

nent services. The algorithm then extracts the inputs and constraints attached to the given service-node and use them as the second and fifth arguments respectively to *DefineSvc* procedure while leaving its last argument blank (depicted as ∅) as there are no component service definitions to be included in the given service's *where* clause. During execution of the OBJECTIVE LUCID program so generated, component service name, which serves as the first argument to *DefineSvc*, is used to invoke the free function defined for the service in the JAVA segment with the input values extracted from its LUCID expression's local context (using # operator) passed as arguments to the function call and the outputs obtained by processing them returned as an object from the JAVA segment.

4 SOLUTION EVALUATION

To assess the operational aspect of our solution, we focus our evaluation process on ensuring that the composition and translation applications that we developed based on these designs function absolutely in accordance with them and, hence, achieve each of the research goals defined on the operational level. We meticulously analyzed every step of the algorithms and prepared exhaustive lists of all their constituent operations/tasks together with all the essential conditions to be met and properties to be exhibited at every step. We divided the composition and translation processes into various stages each of which has a specific goal to be achieved by performing a well-specified series of tasks. For example, the forward expansion stage of the composition process aims at generating a valid search graph while the backward search stage is focused on extracting valid solution plan sets from that search graph. Each separate modular task can then be more effectively specified and then tested. We manually designed test cases, including atomic and composite services, repositories, and composition requests required as input for each of them, to ensure that each of the required operations, conditions, and properties are properly incorporated into the implemented solution and that the composition and translation applications function as per their designs both individually and as a unified process. This thorough analysis and careful design enabled us to design 76 different unit tests.

5 CONCLUSION

Owing to advantages such as clarity of structure, reusability of components, broader options for users and liberty to specialize for providers, composite web services have been extensively researched over the past two decades. Yet, from a thorough review of the literature available on the studies undertaken in the field so far, we gather that most of these studies fail to acknowledge that every service has a limited context in which it can successfully perform its tasks, the boundaries of which are defined by the internal constraints placed on the service by its providers. When used as part of a composition, the restricted context-spaces of all such component services together define the contextual boundaries of the composite service as a unit, which makes internal constraints an influential factor for composite service functionality. However, due to the limited exposure received by this aspect of web service composition, no systems (as per our knowledge) have yet been proposed to cater to the specific verification and validation of internal constraints imposed on components of a composite service.

Based on the concept of context found in intensional logic together with the definition proposed by (Laleh, 2018), the execution context of a web service can be considered as a set of all its input parameters paired with the values assigned to them at the time of the service call. In case of composite services, many of these contextual parameters for component services could get their values assigned dynamically as the composite service is being executed. Therefore, in order to check if the restrictions placed on such variables (i.e., the internal constraints) are satisfied, any verification solution proposed would need to

either actually execute the composite service or else simulate its execution.

In an attempt to address the above problems related to web service composition, in this paper, we propose the use of GIPSY as a simulation/execution-based environment for verification and validation of constraint- and context-aware composite web services. Since GIPSY is a system dedicated to the compilation and execution of LUCID programs, it requires the composite services under examination to be translated into programs written in some LUCID dialect. Therefore, as part of our proposed solution, we design and implement in JAVA an automated and extensible translator framework that allows modules for translating composite services into OBJECTIVE LUCID.

In order to assess the extent to which these two processes fulfill their design goals, we prepare exhaustive lists of all their constituent operations together with the other required conditions to be met by each of their algorithms and perform tests on them individually as well as all of them combined as a composition or translation process. Taking our scope and time restrictions under consideration, the meticulous study that we have conducted on the composition methodology and the LUCID/GIPSY model, we can conclude that we have been able to effectively evaluate our composition and translation solutions and have found them to be capable of fulfilling all their design objectives.

6 LIMITATIONS AND FUTURE WORK

During the course of this research, we discovered several features that can be incorporated into our current verification solution in order to make it more comprehensive, maintainable, efficient, robust, reliable and versatile while improving the quality of the compositions that it generates and validates. The current user interface for the service composition application can be enhanced to assist the user in applying customized termination conditions on the forward expansion process for increased control over the processing duration. For instance, growth of a search graph can be stopped once a certain number of possible solutions are likely to have been obtained, a certain number of layers have been constructed or a certain amount of time has been consumed. Although we allow service requesters to specify their constraints and expected QoS features as part of a composition request for the sake of completeness, we do not implement any mechanisms as of now that would enable requester constraints or QoS features to be included in a solution plan. While our composition application allows composite services to be used as components in other compositions, our translator framework still assumes the components of its source composite services to be atomic in nature. We plan to improve the design of our existing translator modules so as to be able to represent composite component services in the translations generated. We currently use a scenario-based testing approach which does not represent an absolute proof of absence of unexpected or faulty behavior. We are working on the design of a formal model of the OBJECTIVE LUCID translation model followed by development of a correctness proof.

REFERENCES

Ashcroft, E. A., Faustini, A. A., Jagannathan, R., and Wadge, W. W. (1995). *Multidimensional Programming*. Oxford University Press, London.

Chen, M., Tan, T. H., Sun, J., Liu, Y., and Dong, J. S. (2014). VeriWS: A tool for verification of combined functional and non-functional requirements of web service composition. In *Proc. of the 36th Int'l Conf. on Softw. Eng.*, pages 564–567, New York, NY, USA. ACM.

Cheng, J., Liu, C., Zhou, M., Zeng, Q., and Yla-Jaaski, A. (2015). Automatic composition of semantic web services based on fuzzy predicate petri nets. *IEEE Trans. on Automation Sc. and Eng.*, 12(2):680–689.

Dechsupa, C., Vatanawood, W., and Thongtak, A. (2016). Formal verification of web service orchestration using colored petri net. In *Int'l MultiConf. of Eng. and Comp. Sc.*, volume 1.

Huynh, K. T., Quan, T. T., and Bui, T. H. (2015). Fast and formalized: Heuristics-based on-the-fly web service composition and verification. In *2nd Nat. Found. for Sc. and Techn. Devel. Conf. on Inf. and Comp. Sc.*, pages 174–179.

Juan, S. and Hao, W. (2012). Performance analysis for web service composition based on queueing petri net. In *Int't Conf. on Softw. Eng. and Service Sc.*, pages 501–504. IEEE.

Laleh, T. (2018). *Constraint Verification in Web Service Composition*. PhD thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada.

Laleh, T., Paquet, J., Mokhov, S., and Yan, Y. (2018). Constraint verification failure recovery in web service composition. *Future Generation Computer Systems*, 89:387 – 401.

Laleh, T., Paquet, J., Mokhov, S. A., and Yan, Y. (2017). Predictive failure recovery in constraint-aware web service composition. In *Proc. of the 7th Int'l Conf. on Cloud Computing and Services Sc.*, pages 241–252. SciTePress.

Mokhov, S. and Paquet, J. (2005). Objective Lucid – first step in object-oriented intensional programming in the

GIPSY. In *Proceedings of the 2005 International Conference on Programming Languages and Compilers (PLC 2005)*, pages 22–28. CSREA Press.

Mokhov, S. A. (2005). Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY. Master's thesis, Dept. of Comp. Sc. & Softw. Eng.

Nagamouttou, D., Egambaram, I., Krishnan, M., and Narasingam, P. (2015). A verification strategy for web services composition using enhanced stacked automata model. *SpringerPlus*, 4(1):98.

Paquet, J. (2009). Distributed eductive execution of hybrid intensional programs. In *Proc. of the 33rd Ann. IEEE Int'l Comp. Softw. and Appl. Conf.*, pages 218–224. IEEE Computer Society.

Shkarupylo, V. (2016). A simulation-driven approach for composite web services validation. In *Central European Conf. on Inf. and Intel. Systs.*, page 227. Faculty of Organization and Informatics Varazdin.

Siala, F., Ait-Sadoune, I., and Ghedira, K. (2014). A multi-agent based approach for composite web services simulation. In *Int'l Conf. on Model and Data Eng.*, pages 65–76. Springer.

T. Laleh, J. P., Mokhov, S., and Yan, Y. (2017). Constraint adaptation in web service composition. In *2017 IEEE International Conference on Services Computing (SCC)*, pages 156–163.

Wadge, W. W. and Ashcroft, E. A. (1985). *Lucid, the Dataflow Programming Language*. Academic Press, London.

Wang, P., Ding, Z., Jiang, C., and Zhou, M. (2014). Constraint-aware approach to web service composition. *IEEE Trans. on Systems, Man, and Cybernetics Syst/*, 44(6):770–784.