EXECUTION/SIMULATION OF CONTEXT/CONSTRAINT-AWARE COMPOSITE SERVICES USING GIPSY

Jyotsana Gupta

A thesis

in

The Department

of

Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science (Computer Science) at
Concordia University
Montréal, Québec, Canada

July 2019

CONCORDIA UNIVERSITY

School of Graduate Studies

This is to certify that the thesis prepared

By: **Jyotsana Gupta**

Entitled: **Execution/Simulation of Context/Constraint-aware Composite Services using GIPSY**

and submitted in partial fulfillment of the requirements for the degree of

Master of Computer Science (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining committee:

______________________________ Chair
Dr. Denis Pankratov

______________________________ Examiner
Dr. Yann-Gaël Guéhéneuc

______________________________ Examiner
Dr. Nikolaos Tsantalis

______________________________ Supervisor
Dr. Joey Paquet

Approved by ______________________________
Chair of Department or Graduate Program Director

__________ 20 _____ ______________________________
Dr. Amir Asif, Dean
Gina Cody School of Engineering and Computer Science

Abstract

Execution/Simulation of Context/Constraint-aware Composite Services using GIPSY

Jyotsana Gupta

For fulfilling a complex requirement comprising of several sub-tasks, a composition of simple web services, each of which is dedicated to performing a specific sub-task involved, proves to be a more competent solution in comparison to an equivalent atomic web service. Owing to advantages such as re-usability of components, broader options for composition requesters and liberty to specialize for component providers, for over two decades now, composite services have been extensively researched to the point of being perfected in many aspects. Yet, most of the studies undertaken in this field fail to acknowledge that every web 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 them, 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, in this thesis, we propose a multi-faceted solution capable of not only automatically constructing context-aware composite web services with their internal constraints positioned for optimum resource-utilization but also of validating the generated compositions using the General Intensional Programming SYstem (GIPSY) as a time- and cost-efficient simulation/execution environment.

Acknowledgments

My entire time at Concordia University has been a tremendous learning experience, the credit for which is shared by many. First and foremost, I would like to express my deepest gratitude to the university itself for accepting me as a graduate student and providing me with the opportunity to learn from some of the best instructors in the field of Computer Science. I cannot thank Dr. Joey Paquet enough for being an incredibly supportive, patient and kind mentor for a difficult student like me during the entire course of my studies here and for his exceptional guidance and constant encouragement throughout this research. I would also like to thank him, the university and NSERC for providing the funding for this research.

I am profoundly grateful to Dr. Serguei Mokhov, Dr. Touraj Laleh and Alexandre Simard not only for all their excellent research, which forms the basis of this thesis, but also for sharing their invaluable knowledge on so many topics, simple and complex, that helped me better understand my research problems and design an effective solution.

I have been extremely fortunate in being a part of a wonderful research team where there is never a lack of motivation, encouragement, support, learning or laughter, and I would like to thank everybody responsible for it – Joey, Serguei, Touraj, Alex, Eric, Bernie, Jashanjot and Peyman.

I have come a long way since the day I first started preparing for my graduate program applications, and the journey, especially the initial steps, would have proven to be much more daunting and might never even have happened without the selfless help and encouragement of my old friend, Nikhil, his lovely parents and my best friends, Nupur and Kritika.

No words can express my gratitude and love for my amazing family. The unconditional love, unwavering faith and endless patience that my brother, parents and grandparents have blessed me with has made me who I am today. I dedicate this thesis and all my work that has gone into this research to them, particularly my mother and brother who gave their all for making my dreams come true.

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

Chapter 1

Introduction

We begin this chapter with a detailed description of the problem domain and the specific issues that we address in this thesis. Then, we discuss the contributions and define the scope of our research. Finally, we provide a brief overview of our research methodology followed by an introduction to the chapters that explore it elaborately.

1.1 Problem Analysis

Web services or, simply, *services*, are independent, self-describing, modular programs that can be published, searched for, invoked and executed via the World Wide Web. These programs are hosted on computers known as servers and are accessible for use from other computers known as clients provided that both the server and the clients are connected to the Internet. In order to make their services discoverable, *service providers* usually publish them on some globally-available *service registry*, such as the one defined by the Universal Description, Discovery and Integration (UDDI) specifications. Clients, also known as *service requesters* or *service users*, looking for a ready-made application can search through such registries for locating the web service that best suits their requirements. A registry entry is responsible for providing introductory information about a service, such as, its name, type, publisher, language, operating system and a link to its detailed description. A service description, on the other hand, aims to describe the service interface, i.e., the type of inputs accepted and outputs produced by the service, protocols and messages to be used for communicating with it, operations/functions that it can perform and its location. Using the information from the registry and the description, a requester can make a well-informed decision about service selection and make a remote call to the most suitable candidate over the Web to perform the required task [1, 2]. The relationships between the various entities involved in the global interpretation of the web service domain as discussed here have been depicted in Figure 1.

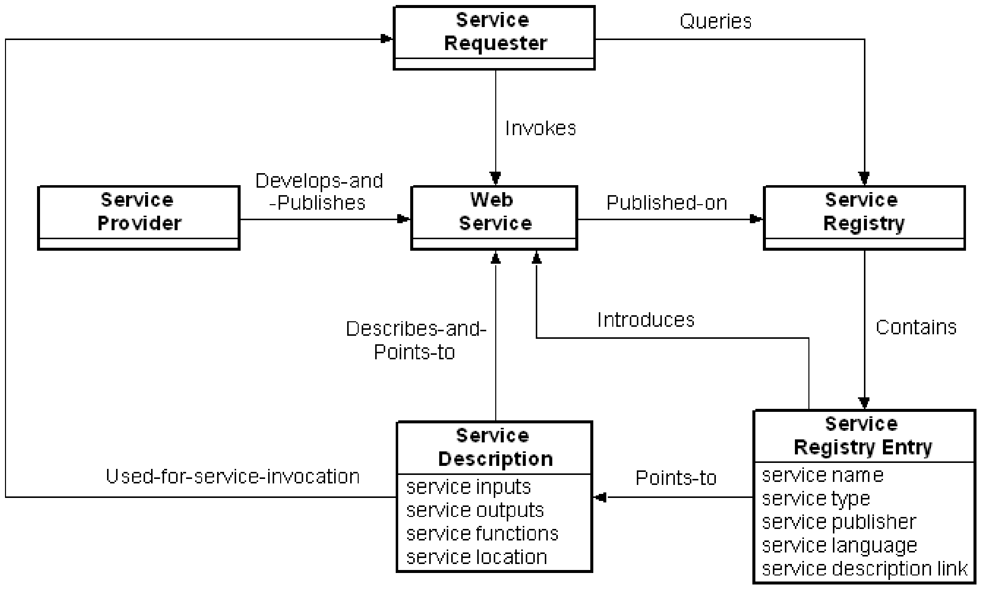


Figure 1: Web Service Domain Model

In order to enhance their clarity and re-usability, web services are usually designed to perform very simple and specific tasks. For instance, consider a web service that processes credit card payments. Such a service would accept credit card details and payment amount as inputs and produce payment status (complete/declined) as output. If such a service is made available as an independent unit, it could be useful for several different domains, such as online shopping stores, student fee payment portals and checkout counters in grocery stores. However, if the same service provider was instead to offer a larger and more complex atomic shopping service (as the one depicted in Figure 2) that could display a product catalog, accept a customer's order, process credit card payment and initiate shipment of the ordered goods, the scope of their potential clients would be significantly reduced because such a shopping service would only be suitable for online shopping stores. Additionally, this shopping service would be leased by a client only if all of its components – Catalog, Order, Payment and Shipment – perfectly fit the client's requirements. If even one of the components would fail to fulfill the required task, the entire service would be rejected. For example, if the service requester needs the service to be able to ship products across the entire North American region, but the service is unable to process Mexican addresses, the whole shopping service would be discarded as unsuitable. That is why it is often difficult to find a ready-made monolithic service on the Web that could perfectly fulfill a complex set of requirements. Therefore, to accomplish such complex tasks, simpler web services (called *component services*) are selected for each sub-task and composed together in the form of a workflow to build what is known as a *composite web service.* These are customized services assembled and arranged according to each target user's particular requirements. Not only does this process of *web service composition* widen the client base for each service provider, it also opens up a wide variety of service options to choose from for each sub-task for the service users, thereby benefiting both the parties involved in the transaction.

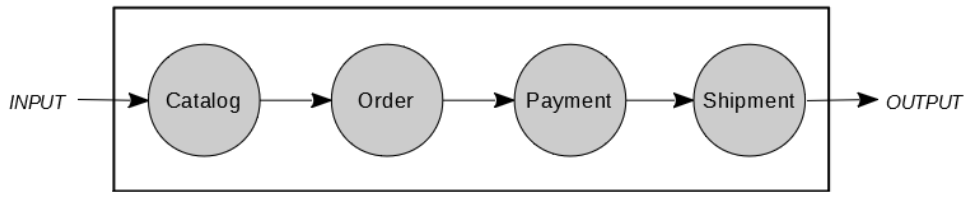


Figure 2: Online Shopping Service Components

Besides providing clarity, re-usability and broader business options, "composable" web services allow service providers to focus on refining the applications that they specialize in while employing external services from other providers to assist with the supporting activities, thereby aiding production of superior quality services in every field. For instance, an online store can dedicate its resources to building a richer product catalog service if they outsource the job of processing payments to an external service provider that specializes in banking applications, thereby creating a superior composite shopping service (similar to the one depicted in Figure 2) resulting in higher customer satisfaction.

Owing to such advantages, among others, the area of web service composition has been extensively researched for over two decades now. 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 equal consideration. For instance, almost all of the existing research on web service composition tends to overlook the fact that no web service has a universal aptitude. Every service has a limited context (albeit possibly wide) in which it can successfully perform its tasks. Such limitations are defined by the service providers and are termed as *service constraints* or *internal constraints*. For example, while one credit card payment service (say, W_1) might be able to process both Visa and Master cards (internal constraint: $CreditCardBrand \in \{Visa, Master\}$), another service (say, W_2) might be capable of processing only Visa cards (internal constraint: $CreditCardBrand = Visa$). Similarly, a shipment service (say, W_3) might be capable of delivering products only to addresses within Canada (internal constraint: $ShippingAddress \in \{Canada\}$). Also, it would be nearly impossible to find a web service that could successfully process every existing credit card brand or one that could deliver products to every country in the world. Now, if the shipment service W_3 and the Visa card payment service W_2 mentioned here were to be used as components of the composite online shopping service depicted in Figure 2, the shopping service as a unit would be constrained to $\{CreditCardBrand = Visa \cap ShippingAddress \in \{Canada\}\}$, i.e., it would 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 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. To the best of our knowledge, internal constraints in the context of web service composition have only been discussed in elaborate detail by Wang, Ding, Jiang, and Zhou in [3], followed by Laleh, Paquet, Mokhov, and Yan in [4, 5, 6, 7, 8].

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, functional requirements or Linear Temporal Logic (LTL) properties. The focus of researchers working on QoS constraint verification has been on ensuring that services maintain certain pre-defined levels of QoS standards, i.e., they offer optimum values for one or more QoS features such as cost, availability, response-time and reliability [9, 10, 11, 12, 13, 14, 15]. Meanwhile, the systems proposed for verification of functional requirements have been dedicated to confirming that services properly perform the tasks claimed by their description [9, 13, 14, 15, 16, 17, 18, 19]. For instance, while testing a credit card payment service, such systems would aim to ensure that given the credit card details and the amount to be paid, the service would generate a receipt if the card details are valid and the payment amount fits within the credit limit. What these systems fail to consider is that not every brand of credit cards can be processed by a single service even if the card details might be valid and the credit limit might allow the payment, in which case, given the details of a credit card that is not recognized by the service, the result should be some appropriate error message. However, such scenarios are not taken into account by the systems proposed so far. Similar is the case with LTL-validation solutions, whose primary focus is on ensuring that the components of a composite service are executed in the correct order [14, 20]. For example, in case of the online shopping service depicted in Figure 2, an LTL-verification solution would mainly be concerned about confirming that the Shipment service is executed only after the Payment service has successfully finished its processing.

The internal service constraints that we aim to verify in this thesis are different from those discussed in the research works cited above. These are, as stated earlier, restrictions imposed on the context in which a service can be executed. We borrow the concept of the *execution context* of a web service from [4] and consider it to be an aggregation of all the information that could affect the execution of the service. According to this definition, each element of an execution context is a name-value pair. Based on that, the context of a service is considered to be the set of all its input parameters and the values that are assigned to them at the time of the service call. While for an atomic service these parameters are specified by the service provider as part of its definition, for a composite service – created in response to a composition request, the execution context is viewed as a set of the input parameters specified as part of the request for execution of the entire composite service, i.e., the input parameters whose values can be supplied by the customer for whom the composite service is intended (see Section 3.1 for details on composition request). For example, consider the composite shopping service depicted in Figure 2 and its component details provided in Table 1. In the table, we list the values that might get assigned to the component services' input and output parameters for a sample run of the shopping service. For this sample run, the context of the atomic services would be:

- **Service W_1:** {*ProductName : StudyTable*}
- **Service W_2:** {*ProductNumber : ST1234, ProductPrice : 75.00*}
- **Service W_3:** {*OrderNumber : ORD1234, PaymentAmount : 82.50, CreditCardBrand : Visa, CreditCardNumber : CCVS56789*}
- **Service W_4:** {*PaymentStatus : Complete, ProductWeight : 45, ShippingAddress : Canada*}

while the context of the composite shopping service would be: {*ProductName : StudyTable, CreditCardBrand : Visa, CreditCardNumber : CCVS56789, ShippingAddress : Canada*}.

For any composite service, many of the context parameters/variables 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 need to either actually execute the composite service or else simulate its execution. For example, consider the online shopping service depicted

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}	$C_{41} = \{ProductWeight <= 50\}$ $C_{42} = \{ShippingAddress = Canada\}$

in Figure 2 and detailed in Table 1. Suppose, the Shipment service provider imposes a 50-pound weight-limit on each package that can be 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.

Moreover, according to the research conducted by Khodadadi in [21], many of the services available on the web are just shell services; i.e. while their descriptions are listed on service registries, the services themselves are either outdated or completely non-functional. Even for functional services, descriptions are often found to be unsynchronized with the latest behavior of their corresponding services. In order to protect potential clients from incorporating such malfunctioning/misinforming services into their compositions, at least a basic execution of each candidate composite service would be required before it could be approved for deployment, which further necessitates building of an execution-based verification system for constraint-aware composite web services.

Additionally, since internal service constraints are, in essence, limitations placed on a service's execution context, a system meant for verifying such constraints must be able to interpret the concept of execution context. However, despite being so closely related to each other, the concepts of "service execution context" and "internal service constraints" seldom appear together in any existing research work, as per our literature review. This exhibits a significant gap in the research being conducted on web service composition.

In an attempt to address the gaps and problems related to web service composition that have been discussed in this section, in this thesis, we propose the use of the General Intensional Programming SYstem (GIPSY) [22, 23, 24] as a simulation/execution-based environment for verification and validation of constraint- and context-aware composite web services. A detailed discussion on GIPSY and its related background can be found in Chapter 2.

1.2 Thesis Objectives and Motivation

An online store, such as Amazon, needs a shopping service that is capable of displaying a product catalog to its customers, accepting the customers' orders, processing their credit card payments and initiating shipment of the ordered goods. Such a store effectively requires a composition engine for such a service, specifying the inputs that its customers would be able to provide and the outputs that they would expect in return. Based on the composition request received and the set of atomic services available to perform the required tasks, the composition engine composes an online shopping service same as the one depicted in Figure 2. However, as discussed in Section 1.1, atomic services tend to have certain 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 upto 50 lbs and need to get their purchased goods delivered within Canada. Therefore, it is essential for the composition engine to take the internal constraints of the Payment and Shipment services into account while assembling the shopping service. At this point, we can deduce the ***first objective*** of our research: ***"To have an operational service composition mechanism that takes into account the execution context of services as well as the restrictions/constraints imposed on them. Based on the requested inputs-outputs and available atomic services, this mechanism should be able to generate one or more solutions, if possible, to any valid composition request."*** For the purpose of this thesis, we assume that service requesters neither have any constraints of their own nor do they object to the internal constraints placed on their requested composite services. We plan to incorporate the requester/user constraints in future extensions of this research (see Section 6.2).

Once the composition engine assembles a suitable composite shopping service, it needs to subject the service to some basic behavioral competency tests, such as the ones listed below, before it can be proposed as a practicable solution to the service requester:

- Component services should accept inputs (whether provided by the user or produced by other component services within the composition) and generate outputs in accordance with their public interfaces.

- Component services should function cohesively as a single unit, i.e., for a valid set of user inputs fed to the composite service, the requested outputs' values should fall within the expected range.

- Internal constraints, wherever applicable, should be enforced properly. As soon as a contextual value is found to be in violation of a restriction, the execution of the composite service should be halted and appropriate error messages generated.

- Irrespective of the kind of combination of component services – sequential, parallel, split or join, the composite service should be able to complete its execution correctly within a valid context space or halt appropriately in case of constraint-violation.

- Each of the component services should be accessible and functional. If any inaccessible or malfunctioning services are detected, the execution of the composite service should be halted, appropriate error messages generated and the composition discarded.

However, as mentioned in Section 1.1, 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, in this situation, the composition engine 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 solution as a feasible one, following which it might even enter into a binding contract or lease with the providers of each of the component services involved.

Now, 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. However (for example), the online shopping store may not be allowed to accept Master cards because of an agreement with the Visa company. 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 because of the alternate behavior of the Payment service. 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 would further 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), which no longer holds any utility for it, or even face legal consequences if the Visa company chooses to view any orders placed using Master cards as a breach of agreement on the store's part. 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. Clearly, it is of utmost significance that composite services be tested thoroughly for correct behavior within, along, and outside the boundaries of their defined execution context before being deployed for public use. This incentivizes the ***second objective*** of this thesis: ***"To design and implement a verification and validation system for context- and constraint-aware composite web services"***.

As explained in Section 1.1, 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 need to 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. Additionally, Section 1.1 tries to draw attention to the fact that many service descriptions published on web registries are not linked to genuinely functional services, which further necessitates execution-based testing of services before deployment in order to avoid serious consequences as the ones discussed in the previous paragraph. Hence, the ***third objective*** of our research is: ***"To make our verification system capable of simulating as well as executing context- and constraint-aware composite web 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. Consequently, and with the intent of adding more practical value to this verification system, we identify the ***fourth objective*** of our research to be: ***"To make the simulation and execution of context- and constraint-aware composite web services time- and cost-efficient"***. 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.

1.3 Thesis Contributions

In the process of accomplishing the goals identified in Section 1.2, we primarily endeavor to make the following contributions to the research on web service composition and verification:

1 ***A generic, optimized, operational constraint-aware service composition mechanism:*** The primary goal of this thesis is to provide a mechanism for the simulation/execution of internally-constrained composition solutions generated in response to a composition request in order to verify that their behavior is in accordance with the requester's expectations. However, before we can verify their behavior, we need to be able to generate such compositions. As mentioned in Section 1.1, only a few teams of researchers have studied composition of internally-constrained services in sufficient detail so far. We base this thesis on the research work conducted by Laleh et al. [4, 5, 6, 7, 8] because they not only provide a formal model for constraint-aware composite services but also introduce a novel constraint-adjustment technique in their service composition process that significantly reduces the number of rollbacks required in case a composite service fails during execution due to a constraint-violation, which enhances the time- and cost-efficiency of the simulation/execution process.

As our first contribution, we design the algorithm for composition plan construction missing from Laleh's set of composition algorithms and optimize their other algorithms to increase the number of alternative solutions produced, to minimize the processing effort spent on invalid or redundant components and unnecessary validation checks and to prevent errors resulting from faulty compositions. All these optimizations are reflected in our implementation of these algorithms, which we have designed as an independent and generic application, augmented with an extensible multi-modal input system and capable of composing constraint-aware solutions for any valid composition request and set of available atomic services. Complete details regarding this contribution along with the necessary explanation on the relevant features of Laleh's research have been provided in Chapter 3.

2 ***An inherently-concurrent dataflow execution model for constraint-aware composite services:*** As stated at the end of Section 1.1, we propose the use of GIPSY as the simulation/execution environment for composite web services. Since GIPSY is a system dedicated to the compilation and execution of LUCID programs, composite services intended for execution on GIPSY need to be translated into LUCID programs. Now, LUCID being a dataflow programming language, programs written in it are, at their very core, formalized textual representations of dataflow networks. Therefore, when executed, a LUCID program gets transformed into a virtual dataflow network of parallely-processing components known as filters. Considering that, if a LUCID program were to represent a constraint-aware composite service, its corresponding dataflow network would be composed of concurrently-executing component service filters enveloped in wrappers serving as internal-constraint-verification layers.

While, on the one hand, this parallel processing of component services would serve to reduce the response-time of most composite services, on the other hand, the fact that this concurrency is an inherent property of the LUCID execution model and does not require the programmer to launch, synchronize or manage any threads would eliminate the possibility of errors resulting from thread-mismanagement, thereby making the LUCID/GIPSY solution for composite service verification more robust.

Therefore, as our second contribution, we study, formally define the elements of, and propose the use of this dataflow model for the simulation/execution of constraint-aware composite services in Chapter 2 while establishing its correspondence with the layered model of service composition defined by Laleh et al. (see Contribution **1**) in Chapters 2 and 4.

3 ***GIPSY as an efficient, context- and constraint-aware, simulation- and execution-based composite service verification system:*** There are several benefits of using a Lucid/GIPSY combination solution for the simulation/execution of composite web services. Being an intensional programming language, Lucid, unlike other existing composition/verification systems, enables effortless incorporation of contextual elements in its programs while its "*whenever*" construct allows for a clear and easy definition of service constraints. Meanwhile, GIPSY, being an execution environment for Lucid, conveniently transforms the programmatic version of a composite service into a context- and constraint-aware dataflow network of component services whose inherent concurrency and virtual nature result in a minimal and efficient consumption of resources (see Contribution **2**). In addition to that, GIPSY's eductive, demand-driven approach towards execution together with its warehouse unit capable of storing and being queried for execution results paired with the specific context in which they were achieved significantly reduce the overall time, effort and cost spent on simulation/execution of composite services.

Additionally, in Section 1.1, we examine the need for both simulation and execution capabilities in a verification system for composite web services and mention that none of the existing research works offer such a dual-mode system for constraint-aware services. However, Objective Lucid – a Lucid dialect composed of Java and Lucid constructs – makes it possible to both simulate (by using Java methods to emulate service definitions) as well as execute (by replacing mock definitions with links to real service implementations) composite services effortlessly within the same system.

Consequently, as our third contribution, we put forth a proposal for the use of the Lucid/GIPSY system as a composite service verification solution capable of testing if the internal constraints placed on component services are correctly verified at their optimal locations within a composition plan as defined by Laleh's unique constraint-adjustment technique (discussed in Chapter 3) and of providing the composite service execution statistics required to ensure that no demands are generated for component services guarded by internal constraints that fail verification, make certain that results for duplicate demands are fetched from the GIPSY warehouse instead of being computed explicitly each time and, ultimately, assess the improvement in time- and cost-efficiency achieved for simulation/execution of composite services. Additionally, we examine those features of the GIPSY architecture that are relevant to its use as a verification system, present an elaborate study of the solution background including all related concepts and explore the benefits of using the proposed solution in greater detail. Furthermore, we also introduce a more comprehensive version of "context" as defined by the intensional branch of mathematical logic into the domain of composite web services. Further and more dedicated discussions on this contribution together with its applicable limitations and evaluation can be found in Chapters 2, 4 and 5.

4 ***An automated composite service model translation framework:*** We briefly discuss our rationale behind using Laleh's service composition technique, LUCID's dataflow execution model and GIPSY's simulation/execution environment in Contributions **1**, **2** and **3**. In order to exploit the complete potential of these tools and techniques, we need a translator capable of translating the layered composite services generated by our composition mechanism into OBJECTIVE LUCID programs that could be simulated/executed on GIPSY. However, an isolated translator with a single target language would be a highly restrictive and rigid structure, which would require major design changes for accommodating any future translators. Therefore, as a more flexible and maintainable solution, we design and implement a translator framework capable of allowing modular plugging-in and -out of different translator programs, as required. This framework can accept a layered composite service as input through an extensible multi-modal input system and translate it into any of the target models, such as, LUCID's dataflow execution model, for which a translator module is available, thereby allowing us to leverage the unique qualities of each of the target models/languages for enhancing the overall worth of our verification system. We explain the design, implementation and other relevant features of the translator framework in Chapter 4.

1.4 Thesis Scope

The primary objective of this thesis is to introduce a system that can be used for the verification and validation of context- and internal-constraint-aware composite web services. Since this issue has not been addressed in any research conducted so far, it was our responsibility to examine all the intricacies of this problem and incorporate ways to handle them in our proposed solution. With the limited time and resources available to us, addressing a wide range of requirements in sufficient detail would have been difficult to achieve and could have resulted in an inadequate overall solution. Therefore, for the purpose of this thesis, we decided to focus our efforts on accomplishing a limited yet clearly defined set of goals, as listed in Section 1.2, to the best of our abilities.

The issues or aspects related to web service composition and verification that are outside the scope of this thesis have been briefly described below:

- While verification of the constraints imposed by service providers is the focal point of our research, we do not take into consideration constraints imposed by composite service requesters or restrictions from entities external to the web service domain, such as, government policies. However, these constraint categories are planned to be included into our system in future extensions to this research.

- Since we base our research on Laleh's planning-graph approach to service composition, its associated limitations are also inherited by our solution. Consequently, any composite service that we generate or verify cannot contain loops or multiple occurrences of the same component service. Also, there cannot be any uncertainty in the execution of component services, i.e., each component of a composite service must be executed for the composite service's execution to be completed [4].

- Although Laleh's research offers an algorithm for merging different alternative solutions/plans generated for a given composition request into a package with alternative plans to chose from in case one plan fails during execution [4, 5], we restrict our system to the composition and verification of a single plan at a time in order to avoid complicating our prototype system at such an early stage. We plan to integrate the solution package into our system as part of a future work.

1.5 Research Methodology

The general procedure that we follow for the verification of internally-constrained context-aware composite web services has been depicted in Figure 3. According to the procedure:

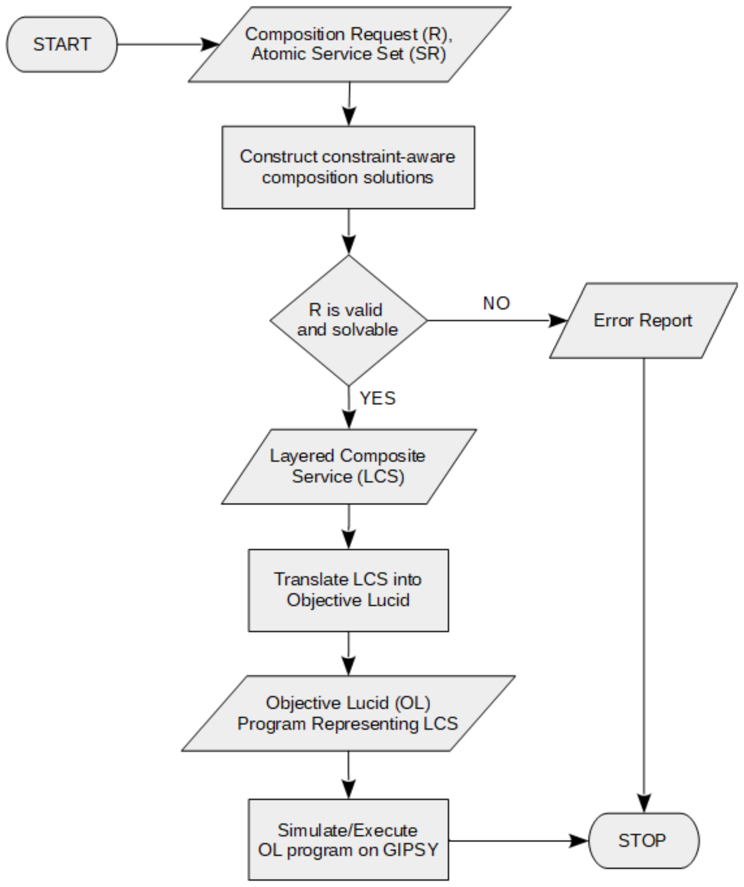

START
Composition Request (R), Atomic Service Set (SR)
Construct constraint-aware composition solutions
R is valid and solvable
NO
Error Report
YES
Layered Composite Service (LCS)
Translate LCS into Objective Lucid
Objective Lucid (OL) Program Representing LCS
Simulate/Execute OL program on GIPSY
STOP

Figure 3: Composite Web Service Verification Procedure

- A composition request, specifying the inputs provided by and the outputs required by the service requester, along with a set of atomic services (with or without internal constraints) available for being assembled into a workflow are fed to our service composition mechanism derived from Laleh's composition approach (see Contribution **1**).

- Based on the given composition problem's validity and solvability using the available atomic services, the composition application generates one or more constraint-aware layered composite services as potential solutions to the problem. Invalid or unsolvable problems are reported back to the requester.

- Each of the layered composite services generated (if any) for the request can then be fed to our service translator application, which transforms them into equivalent Objective Lucid programs capable of being executed on GIPSY (see Contribution **4**).

- The Java definitions of component services constituting an Objective Lucid program may either be generated by the translator to emulate the behavior described by their corresponding descriptions or may utilize their actual implementation sourced from their providers. Depending on whether its component definitions are simulated or real, a composite service may be considered as either being simulated or executed respectively by the GIPSY environment (see Contribution **3**).

In order to evaluate our verification system and prove the accomplishment of the goals defined in Section 1.2, we experiment with a range of composition requests and atomic service sets (see Chapter 5), resulting in the construction of a variety of compositions with diverse counts and combinations of component services. All these compositions are translated into Objective Lucid programs by our translator framework so that they could be simulated or executed on GIPSY and their functional and non-functional characteristics could be observed and recorded in different contexts restricted by each of their specific set of internal constraints as part of future extensions to this research. Such observations would help us in performing statistical analysis of composite service behavior – both positive and negative – and estimating the time and cost benefits of using our system for composite service simulation and execution.

1.6 Thesis Outline

We summarize the purpose of each of the following chapters below:

- **Chapter 2** offers an elaborate discussion on all the relevant aspects of the LUCID/GIPSY verification system essential to obtaining a complete understanding of the proposed solution. It also provides a review of other related research conducted in the field, contrasting it with the approach described in this thesis.

- **Chapter 3** explains Laleh's service composition model and method, on which we have based this thesis, along with the improvements made to their composition approach as part of our research. It also describes the architecture, usage and other defining features of our composition application.

- **Chapter 4** is responsible for reinforcing the correspondence between Laleh's layered model and LUCID's dataflow model for composite services, defining the algorithms for translation of a layered composite service into an OBJECTIVE LUCID program and presenting the architecture, usage and other characteristics of our service translation framework.

- **Chapter 5** describes the tests conducted to evaluate our proposed verification solution (with particular emphasis on the service composition and translation units) along with their results and the inferences drawn from them.

- **Chapter 6** gives a conclusion on the thesis and lists the solution limitations and improvements that need to be addressed in the future works.

1.7 Summary

Despite the extensive research conducted up to now in the field of web service composition, we still find a gap when it comes to verification and validation of context- and internal-constraint-aware composite web services. In an attempt to fill this gap, we propose the use of a Lucid/GIPSY combination as a simulation/execution-based solution to the verification problem. Having identified our specific research goals and the methodology to achieve them in this chapter, in the next chapter, we explore the unique characteristics of the Lucid/GIPSY system and compare it with other related research works in order to facilitate a better understanding of our proposed solution and to support our rationale behind employing it for context- and internal-constraint-aware composite service validation.

Chapter 2

Solution Background

In Chapter 1, we propose using a Lucid/GIPSY combination as a simulation/execution-based verification system for context- and internal-constraint-aware composite web services. In this chapter, we explain those fundamental concepts, characteristics and architectural features related to the Lucid programming language and the GIPSY environment that are not only essential to gaining a comprehensive understanding of our proposed solution but are also responsible for giving the solution an edge over the other studies conducted up to date in the field of composite web service verification, simulation and execution, which we examine specifically in our review of the related works later in this chapter.

2.1 Lucid Programming Language

Lucid is an intensional [25] and dataflow [26] programming language whose distinctive program structure, programming constructs and execution model together make it an ideal solution for efficient representation and execution of constraint- and context-aware composite web services. We begin this section with definitions of the fundamental concepts behind these distinguishing features of Lucid, which are followed by an explanation of the features themselves along with the reasons that prove them to be advantageous to the composite web service domain.

2.1.1 Intensional Logic

At the root of intensional programming is intensional logic – a branch of mathematical logic that aims at describing context-dependent entities [27, 28]. It originated as a means to formally describe the meaning of natural languages, keeping under consideration that a sentence may be interpreted differently when used in different situations, at different locations, by different people and so on. In other words, the meaning of a sentence may vary according to the context in which it is used, thereby making it a context-dependent entity. As an example, consider the following expression:

E: The temperature is below the freezing point.

The meaning of the above expression remains uncertain and may attain arbitrarily different values unless we specify the exact date on and the particular city for which the *temperature* being referred to in the expression is recorded. In other words, an unambiguous interpretation of expression E is conditional upon the knowledge of factors *Date* and *City*. Such factors that influence the interpretation/evaluation of an expression are termed as *dimensions* in the intensional branch of mathematical logic. A range of different values may be attained by a dimension depending on the type of information that it represents (see Table 3 for sample dimension values). When each of the dimension names associated with an expression is paired with one of its applicable values, the set of dimension name-value pairs so obtained is said to be one *context* or *possible world* for that expression to be evaluated in. A collection of all such possible worlds in which the expression can be evaluated (potentially, to a unique result in each world) is known as the *context-space* for the expression. Depending on the number of dimensions that a context-space is composed of, it can be termed as being *one-dimensional* or *multi-dimensional*, and, conceptually, it is possible for a context-space to even be infinitely multi-dimensional. The relationship between the contexts of an expression and its calculated values is called the *intension* of the expression whereas the set of all specific value of its intension corresponding to any particular context is called the *extension* of the expression.

Applying the concepts defined in the previous paragraph to expression E, the *intension*

Table 2: Extension for Temperature in Expression E

City / Date	**Montreal**	**Toronto**	**Ottawa**	**...**
01/01/2018	-10	-5	0	...
02/01/2018	-9	-5	-1	...
03/01/2018	-7	-4	-2	...
...	...	...	...	...

Table 3: Intension of Expression E

City / Date	**Montreal**	**Toronto**	**Ottawa**	**...**
01/01/2018	*true*	*true*	*false*	...
02/01/2018	*true*	*true*	*true*	...
03/01/2018	*true*	*true*	*true*	...
...	...	...	...	...

of E would be a function in $(D \times C) \rightarrow B$, where D and C are the sets of values that can be assigned to the *Date* and *City dimensions* respectively and B is the set of *boolean* values that can be attained for each combination of *Date*-*City* values. A sample mapping of this function (i.e. E's *extension*) has been presented in Table 3. E's extension is based on the extension of the temperature records of each given city on each given date (presented in Table 2). Here, the *context* for the first recorded temperature would be: $\{$*Date : 01/01/2018, City : Montreal*$\}$ whereas the *extension* of *temperature* in that *context* would be -10, and the extension of expression E would be *true*. All such contexts listed in the two tables would together constitute the *two-dimensional context-space* for both *temperature* as well as expression E.

2.1.2 Dataflow Networks

The generally accepted semantic model used to describe Lucid programs relies on a dataflow execution model, i.e. upon execution, a Lucid program is interpreted as a dataflow network. Therefore, in order to gain an insight into this execution model, it is essential to first obtain a clear understanding of dataflow networks in general.

A *dataflow network* is a network of components known as *filters* interconnected through links known as *channels*. Each filter in the network is a processing unit representing a function that can transform the data elements flowing through the network from one form to another, i.e. from input to output. Each channel or *transition* in the network refers to a stream connecting two filters via which data elements are transmitted from one filter to another. The distinguishing characteristic of a dataflow network, which is also its primary advantage, is its inherent concurrency – the ability of its components to carry out their computations in parallel. Furthermore, each of these components is a *black box* and, therefore, does not interact with any of the other components in the network other than while receiving inputs from or sending outputs to them. The inner processing of each filter is completely hidden and shielded from the side effects of other filters, thereby allowing the functions to maintain *referential transparency*, i.e. to ensure that the values of their outputs depend entirely on the values of their inputs, which, in turn, implies that for a given set of inputs, a filter produces the same output each time they are processed irrespective of its past results and of the state of the other filters in the network [26, 29].

As an example, consider the dataflow network depicted in Figure 4. The goal of this network is to calculate the range (i.e. the difference between the maximum and the minimum) of the three numbers, *num*1, *num*2 and *num*3, provided as input. The *maximum* and *minimum* filters in the network have three entry points (one for receiving tokens from each of the three input nodes) and one exit point (for placing the calculated output in the data stream directed towards the *difference* filter) each. The *difference* filter, on the other hand, has only two entry points (one for receiving the output of each of the other filters) and one exit point, which supplies the final result to the output node of the network. As soon as data elements are available at all the input points of a filter, it begins computing the result. Since all required inputs for the *maximum* and *minimum* filters are available simultaneously, being black boxes, the two filters can operate concurrently and completely decoupled from each other. Depending on the rate at which these filters compute their respective results,

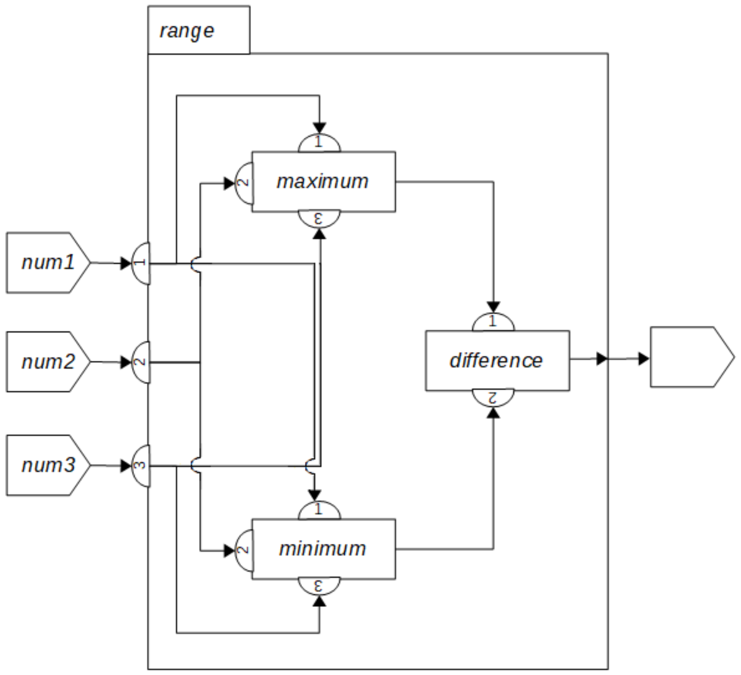


Figure 4: Dataflow Graph for the *range* Program Shown in Listing 2.1

the *difference* filter may receive tokens at its input points at the same or different times. However, owing to the referential transparency property, the results are guaranteed to be the same irrespective of the order in which the two filters complete their processing. Once both inputs are received by the *difference* filter, difference between the maximum and minimum numbers is computed and the result is supplied to the output node [29]. Clearly, it can be seen that a dataflow network, while allowing its constituent filters to operate in a concurrent and asynchronous fashion, manages to exhibit the same consistency of results for any given set of inputs that would be expected of its sequential equivalent, i.e. it combines the benefits of both sequential and concurrent computations.

2.1.3 Lucid Program Structure and Execution

Lucid is a functional programming language, and, therefore, each Lucid program is an expression composed of one or more literals, variables, operators and/or functions, accompanied by definitions of each of the constituent variables and functions themselves as expressions whose components are further defined in the same manner. The purpose of executing these programs is to evaluate the main expression defined in them, which requires evaluation of all its constituent identifiers, which, in turn, depends on the evaluation of their own constituents and so on. Unlike other functional programming languages, however, every expression in LUCID is evaluated in a certain context, which may be comprised of one to infinitely-many dimensions, and can potentially result in a different value in every possible context (as explained in Section 2.1.1). While most mainstream imperative languages need to rely on cumbersome extensional branching to evaluate such expressions in all possible contexts, LUCID, being an intensional language, allows each of the contextual dimensions to be defined as any of its regular variables in a concise and precise manner besides providing operators # and @ for directly extracting values from and specifying values for them respectively. In addition to that, the *whenever* operator supplied by LUCID enables context-dependent conditions to be placed on variable and function definitions, allowing them to be computed only if the conditions evaluate to *true* [26, 25, 28].

As an example, consider the LUCID program shown in Listing 2.1. The objective of the program is to calculate the range of three numbers. Consequently, its main expression is a variable called *range* (defined later in the program) whose value depends on the values of the three numbers in concern, and, therefore, is evaluated in a three-dimensional context where each dimension – g_num1, g_num2 and g_num3 – represents one of the three numbers. At each point of reference in this context space, the *range* expression can potentially evaluate to a different value. In order for it to be computed at a specific point of reference, i.e. to calculate the range of a specific set of numbers, the @ operator must be used. The job of the @ operator is to return the value of its first argument, i.e. the expression, *at* the position, in the appropriate dimension(s), specified by its second argument, i.e. the specific point of context [25]. For the given example, line 1 serves to return the value of *range* based on its definition at a point of reference where the three numbers/dimensions are 10, 12 and 14 respectively. All the definitions related to an expression in a LUCID program are specified as part of the *where* clause associated with it, beginning with the dimensions that define its

Listing 2.1: Lucid Program to Calculate Range of Three Numbers

```
 1  range @.g_num1 10 @.g_num2 12 @.g_num3 14
 2  where
 3      dimension g_num1, g_num2, g_num3;
 4
 5      range = difference (#.l_max, #.l_min)
 6              @.l_max max
 7              @.l_min min
 8              where
 9                  dimension l_max, l_min;
10                  difference (x, y) = x − y;
11              end;
12
13      max = maximum (#.l_num1, #.l_num2, #.l_num3)
14             wvr c_max
15             @.l_num1 #.g_num1
16             @.l_num2 #.g_num2
17             @.l_num3 #.g_num3
18             where
19                 dimension l_num1, l_num2, l_num3;
20                 c_max = #.l_num1 >= 0 and #.l_num2 >= 0 and #.l_num3 >= 0;
21                 maximum(x, y, z) = greater(x, greater(y, z));
22                 greater(a, b) = if a > b then a else b fi ;
23             end;
24
25      min = minimum (#.l_num1, #.l_num2, #.l_num3)
26             wvr c_min
27             @.l_num1 #.g_num1
28             @.l_num2 #.g_num2
29             @.l_num3 #.g_num3
30             where
31                 dimension l_num1, l_num2, l_num3;
32                 c_min = #.l_num1 >= 0 and #.l_num2 >= 0 and #.l_num3 >= 0;
33                 minimum(x, y, z) = lesser(x, lesser (y, z));
34                 lesser (a, b) = if a < b then a else b fi ;
35             end;
36  end
```

context. Each of these dimensions is declared using a *dimension* clause at the top of the body of the *where* clause, indicating that the dimension is new and will be used only inside the enclosing *where* clause [25]. Depending on the location at which it is declared, a dimension may have a global or a local scope in which it can be used. For instance, dimensions *g_num*1, *g_num*2 and *g_num*3 are declared in the outermost *where* clause (line 3) and, therefore, have a global scope, which we indicate by prefixing there names with '*g_*'. On the other hand, dimensions *l_max* and *l_min* (line 9) can be used only within the local scope of the *difference* function and not outside it, which is indicated by their prefix '*l_*'. Dimensions declaration in an expression's *where* clause is followed by definitions of its related constraints, variables and functions. The *range* variable, for instance, is defined as a function called *difference* that accepts two input parameters (line 5) and is, therefore, evaluated in a two-dimensional context (lines 6 - 7) defined by dimensions *l_max* and *l_min*. The input parameters of the function extract their values from its contextual dimensions using the # operator, which is responsible for returning the current value of the dimension specified as its argument [25]. The dimensions of the function, in turn, receive their values from other variables (*max* and *min*) defined later (lines 13 - 23 and 25 - 35) in the program. The computation to be carried out by the *difference* function is defined in its *where* clause (line 10) after its dimensions declaration. The *max* and *min* variables referred to in this definition are defined in a similar manner as the *range* variable except for two major differences. Firstly, values of the local dimensions, *l_num*1, *l_num*2 and *l_num*3, of the *maximum* and *minimum* functions are not computed by executing another function but are retrieved from the global dimensions, *g_num*1 (10), *g_num*2 (12) and *g_num*3 (14) respectively (lines 15 - 17 and 27 - 29). Secondly, these functions have certain constraints placed on them, which are specified using the *wvr* operator – alternate form of *whenever*. Such functions, which form the first argument to the operator, are computed only if the constraints, which form the second argument, evaluate to *true* (lines 14 and 26) [25]. In the given example, all the constraints placed on a function (*maximum* or *minimum*) have been specified as part of its *where* clause (lines 20 and 32) and their result represented as a variable (*c_max* or *c_min*) to serve as the second argument to the *wvr* operator in order to maintain a clear and uniform program structure. Although this example makes use of only the *and* logical operator for combining multiple conditions, Lucid also provides the *or* operator to represent optionality in conditional expressions. Same as the functions which they constrain, these conditional expressions are also evaluated in a specific context, which may be defined as part of a *where* clause associated with them. In the

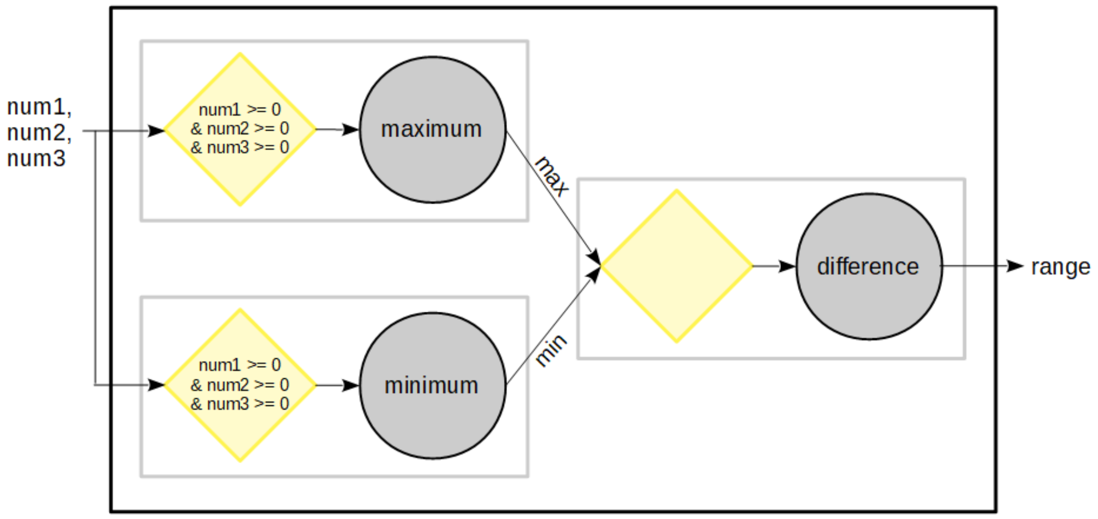


Figure 5: Range Composite Service

given example, however, the constraints and their corresponding functions share the same context, which eliminates the need of a separate *where* clause for the conditions.

Having explained the general functionality, syntax and structure of the program presented in Listing 2.1, we now describe the unique characteristics that the program demonstrates. In the program, contextual values of functions *difference*, *maximum* and *minimum* are also used as inputs/arguments to the respective functions, i.e. inputs to these functions derive their values from the context in which the function is evaluated. The contextual dimensions, in turn, receive as their values the results of certain computations. In other words, in the given program, contextual dimension of a function is the entity that not only serves as context and input to the function but also the data that gets computed, which is not the usual practice. Nevertheless, we present this particular example here because this is the concept that we follow for representing composite web services as LUCID programs – one of the objectives of this thesis. Although this program is not an exact LUCID translation of a composite service, which we discuss in Chapter 4, it still successfully depicts the major features of a composition. It is an approximate representation of a composite service responsible for calculating the range of three numbers, as depicted in Figure 5. The three numbers in concern serve as the inputs that can be provided by the customers of the composite service whereas the range computed by the program acts as the output expected by them. Meanwhile, each of the functions, *difference*, *maximum* and *minimum*, defined in the *where* clause of the program's main expression serve as components of the composite service (depicted as circles in Figure 5) while the conditions placed on them using the *wvr* operator act as internal service constraints (depicted as diamonds in Figure 5). Although we do not make use of Petri nets for representing composite web services in this research, it is still interesting to note here that the constraint diamonds and service circles shown in Figure 5 perform tasks similar to those performed by the *input/pre-condition place* circles and *transition* bars that usually constitute Petri net graphs [12, 19, 30, 31, 32, 33]. As explained in Section 1.1, we define the contextual dimensions of a service, be it atomic or composite, to be the set of all its input parameters. Applying the same definition to the *Range* composite service, the three numbers, which are inputs to not only the composite service itself but also to its *maximum* and *minimum* components, serve as the contextual dimensions for the *range* variable, *maximum* function and *minimum* function in the corresponding LUCID program while their values are used as arguments to the *maximum* and *minimum* functions. Same is the case with the *difference* function as well whose dimensions are the same as the inputs to the *difference* service and their values are used as arguments to the function. However, inputs to the *difference* service are actually the outputs produced by other component services unlike the *maximum* and *minimum* services, which receive their input values directly from the customer. Therefore, in order to compute the input values for the *difference* service, its predecessors, i.e. services that supply its inputs, must first be executed and their outputs computed. This predecessor-successor relationship between component services is represented in the given LUCID program in lines 6 - 7, where the contextual values for the *difference* function are variables that evaluate to the result of computations performed by the *maximum* and *minimum* functions.

For performing the computations required during program execution, LUCID employs a lazy demand-driven approach, known as *eduction*. As already explained, a LUCID program is an expression accompanied by definitions of the identifiers that constitute that expression. In

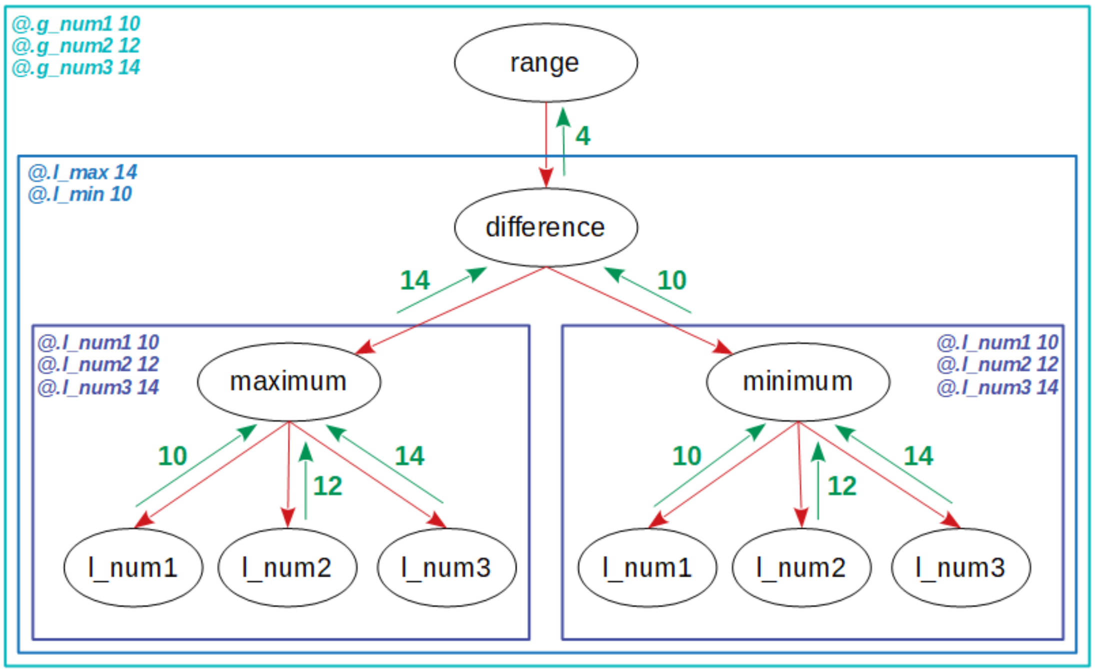


Figure 6: Demand Generation and Computation Tree for the *range* Program

order for this main expression to be evaluated, the values of each of its constituents must be known. The same is true for the evaluation of the expressions that define these constituents as well as for those that define their constituents and so on. According to the eductive model of computation, whenever such an expression needs to be evaluated, a *demand* or request for the value of each of its constituent identifiers in the current context is generated. For each of these demands, the eduction engine inspects the defining expression for its corresponding identifier and generates demands for the constituents of that expression, which, in turn, may lead to generation of further demands and so on. In essence, for each identifier appearing in the main expression of a Lucid program, a tree-like structure (as shown in Figure 6) is incrementally constructed where each node is an identifier-demand whose children are the demands generated for evaluating the identifier's defining expression. Every branch in this tree keeps growing from top to bottom (indicated in red in Figure 6) until its bottom-most node/demand evaluates to a literal value. This value is then propagated upwards to the node's parent in the branch (indicated in green in Figure 6). Once an identifier-demand node receives the values of all its child nodes in a similar fashion, the identifier's definition is applied on them and its value is computed and propagated further upwards in the tree. This process of generation, propagation and consumption of demands and their computed values continues until all the identifiers for the main expression are evaluated and its value can be computed, thereby achieving the goal of the program. A significant advantage of the eduction model is its frugal approach towards computation, which generates a demand for a value only if and when it is required for the computation of another demanded value. Values that are not needed to compute a result are never computed. Such an approach not only saves execution resources but also optimizes the time taken for executing a LUCID program [24, 25].

Another aspect of LUCID that improves its run-time efficiency is its dataflow execution model. As discussed in Section 2.1.2, a LUCID program is a textual representation of a dataflow network and it transforms into that network upon execution. Each definition in a LUCID program can be represented as a filter in its corresponding dataflow network, performing the same computation as specified by the definition itself. The external inputs provided to and the outputs produced by the program serve as the respective inputs and outputs of the network whereas the internal input-output relationships among defining expressions take the form of the network's channels. As an example, consider the *range* program shown in Listing 2.1 and its corresponding dataflow network depicted in Figure 4. For the sake of clarity and simplicity, only the three major defining functions – *maximum*, *minimum* and *difference* – of the program have been represented as filters in the dataflow graph. However, similar graphs can be drawn separately for each of these functions composed of filters representing their constituent definitions in order to elaborately depict their individual operations. For the current example, taking into account the structure of the *range* program, its equivalent dataflow graph and the composite service (shown in Figure 5) that it approximately represents, it can be clearly seen that while the dataflow graph represents the *range* composite service as a whole, each of its constituent filters represents one of its components. Furthermore, as we already know, each filter in a dataflow network is capable of functioning in parallel with the other filters. Therefore, it can be concluded that by translating a composite service into a LUCID program, parallel operation of its component services can be achieved, which can reduce the time required by the service for completing its processing. Moreover, this concurrency of operation is available as an inherent property of LUCID that does not demand any additional programming effort for launching or managing of multiple threads, which, in turn, eliminates the risk of complex errors commonly known to originate from thread mismanagement. In other words, LUCID provides the means of improving the efficiency of composite service execution through inherent concurrency without exposing it to the risks of faulty multi-threading [26, 29].

2.1.4 OBJECTIVE LUCID

Since the origin of LUCID in 1974 [26], several different dialects of the language have been developed each one with its own distinguishing characteristics. Of these variants, GLU (Granular LUCID) was the first one to pair LUCID with a non-intensional programming language, employing LUCID – the intensional language – for specifying the parallel structure of an application and C – the imperative language – for specifying the application's functions, thereby combining the ease of programming in mainstream languages with the efficiency of intensional dataflow languages [34, 35]. From a dataflow perspective, this hybrid language uses LUCID to specify the filters and connecting channels that constitute the dataflow network equivalent of a program while using C to define the operations performed by each of these filters. Another LUCID dialect based on the same hybrid concept is OBJECTIVE LUCID, which replaces C with JAVA as its imperative component, allowing its intensional segment to not only manipulate JAVA objects as first class values but also use JAVA's *dot-notation* for manipulating the members of those objects [36, 37]. In our research, we use OBJECTIVE LUCID for representing composite web services so as to allow them to be simulated/executed on GIPSY (the rationale for which is discussed later in this section).

Each program written in OBJECTIVE LUCID comprises of two code segments: one written in JAVA whose beginning is marked by a *#JAVA* tag and the other in LUCID marked by a

Listing 2.2: OBJECTIVE LUCID Program to Calculate Range of Three Numbers

```
#JAVA
public class ReqComp
{
    private int diff;

    public ReqComp(int diff)
    {
        this.diff = diff;
    }
}

public class Difference
{
    private int x;
    private int y;
    private int diff;

    public Difference(int x, int y)
    {
        this.x = x;
        this.y = y;
        diff = 0;
    }

    public void process()
    {
        diff = x - y;
    }
}

public Difference calcDiff(int x, int y)
{
    Difference oDifference = new Difference(x, y);
    oDifference.process();
    return oDifference;
}

public class Maximum
{
    private int x;
    private int y;
    private int z;
    private int max;

    public Maximum(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        max = 0;
    }

    public void process()
    {
        max = x;
        if (max < y)
            max = y;
        if (max < z)
            max = z;
    }
}

public Maximum calcMax(int x, int y, int z)
{
    Maximum oMaximum = new Maximum(x, y, z);
    oMaximum.process();
    return oMaximum;
}

public class Minimum
{
    private int x;
    private int y;
    private int z;
    private int min;

    public Minimum(int x, int y, int z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
        min = 0;
    }

    public void process()
    {
        min = x;
        if (min > y)
            min = y;
        if (min > z)
            min = z;
    }
}

```

```
public Minimum calcMin(int x, int y, int z)
{
    Minimum oMinimum = new Minimum(x, y, z);
    oMinimum.process();
    return oMinimum;
}

#OBJECTIVELUCID
oRange @.g_num1 10 @.g_num2 12 @.g_num3 14
where
    dimension g_num1, g_num2, g_num3;

    oRange = ReqComp(#.l_diff)
              @. l_diff  oDifference. diff
              where
                  dimension l_diff ;

                  oDifference = calcDiff(#.l_max, #.l_min)
                                @.l_max oMaximum.max
                                @.l_min oMinimum.min
                                where
                                    dimension l_max, l_min;
                                end;

                  oMaximum = calcMax (#.l_num1, #.l_num2, #.l_num3)
                                wvr c_max
                                @.l_num1 #.g_num1
                                @.l_num2 #.g_num2
                                @.l_num3 #.g_num3
                                where
                                    dimension l_num1, l_num2, l_num3;
                                    c_max = #.l_num1 >= 0 and #.l_num2 >= 0 and
                                            #.l_num3 >= 0;
                                end;

                  oMinimum = calcMin (#.l_num1, #.l_num2, #.l_num3)
                                wvr c_min
                                @.l_num1 #.g_num1
                                @.l_num2 #.g_num2
                                @.l_num3 #.g_num3
                                where
                                    dimension l_num1, l_num2, l_num3;
                                    c_min = #.l_num1 >= 0 and #.l_num2 >= 0 and
                                            #.l_num3 >= 0;
                                end;
              end;
end
```

#OBJECTIVELUCID tag. For example, consider the program shown in Listing 2.2, which is an OBJECTIVE LUCID translation of the pure (Indexical) LUCID program to calculate the range of three numbers shown in Listing 2.1. Comparing the LUCID segment of this OBJECTIVE LUCID program with its Indexical LUCID equivalent, three major differences between them can be noticed. Firstly, OBJECTIVE LUCID replaces the declarative definitions of functions *difference*, *maximum* and *minimum* with procedural definitions of JAVA methods *calcDiff* (lines 31 - 36), *calcMax* (lines 63 - 68) and *calcMin* (lines 95 - 100) respectively (specified in the JAVA segment), which can be invoked from the LUCID segment through regular method call statements (lines 112, 119 and 129). Secondly, unlike their declarative counterparts, which return only simple variables, these procedural functions may also return JAVA objects (such as *oDifference*, *oMaximum*, *oMinimum*), which could be composed of one or more data members and/or member functions of different data/return types. Each of these members can be accessed within the LUCID segment using the *dot* operator on the object in concern. For instance, in the given OBJECTIVE LUCID example, the function *calcDiff* creates an object *oDifference* of class *Difference* (line 33), calls its member function *process* (line 34) for computing the difference between its two arguments and storing the result in *oDifference*'s data member *diff* (lines 25 - 28) and finally returns the object (line 35). The computed difference value is then accessed in the LUCID segment using the *dot* operator on object *oDifference* (line 108) returned by *calcDiff* (line 112). The same is the case with the computation and access of the maximum and minimum values in the program. Thirdly, while the final output of the Indexical LUCID program is a simple variable called *range*, which stores the return value of the *difference* function, the output of the OBJECTIVE LUCID program is an object called *oRange* created by invoking the *ReqComp* constructor and comprising of a single data member *diff*, which, in this program, is responsible for holding the return value of the *difference* function. Moreover, as can be inferred from the program, the value of *oRange* is directly dependent only on *ReqComp*, which, in turn, depends on the functions that compute the difference, maximum and minimum values. Therefore, these functions have been moved from the global scope (such as in the Indexical LUCID program)

to the local scope defined by *ReqComp*'s *where* clause in the Objective Lucid program.

Our purpose behind using Objective Lucid and the particular language constructs discussed in the previous paragraph is to build a complete and clear representation of context- and constraint-aware composite services that can be simulated/executed on GIPSY. Since GIPSY is dedicated to the compilation and execution of Lucid programs, it becomes necessary for composite services intended to be executed on it to first be translated into some Lucid dialect. In order for this dialect to be able to represent all the required features of the composite service model that we use (discussed in Chapter 3), it must exhibit certain specific properties, which, based on our knowledge of Lucid variants, is accomplished (without introducing any superfluous characteristics) only by Objective Lucid. As rationale behind this selection, we present a discussion on all such requirements as well as the Objective Lucid constructs that help fulfill them below:

1 **Requirement:** Parameters that act as inputs to a service (whether atomic or composite) or on which constraints are placed should be allowed to serve as contextual dimensions while retaining the capability of being defined, computed and used as regular variables.

  Solution: As explained in Section 2.1.3, Indexical Lucid allows service inputs and constraint features to be defined as contextual dimensions using the *dimension* clause and the @ operator. The same holds true for the Lucid segment of an equivalent Objective Lucid program. Furthermore, in the Java segment of the program, service inputs can be passed and processed as regular function arguments and objects' data members within their corresponding service definitions. Meanwhile, constraint features can be used as regular variables in those Lucid conditional statements that define these service constraints. For instance, inputs *max* and *min* to the *difference* component of the *range* composite service (depicted in Figure 5) are defined both as inputs as well as dimensions, l_max and l_min, for its corresponding function *calcDiff* in Listing 2.2 (lines 112 - 117) while being processed as regular variables as part of their service definition in the Java segment (lines 12 - 36). Similarly, values of l_num1,

l_num2 and l_num3, which serve as shared contextual dimensions for function *calcMax* and constraint c_max (lines 121 - 123 and 125), are conveniently used as regular parts of the constraint's definition (line 126) in the LUCID segment of the program.

2 **Requirement:** Services (whether atomic or composite) should be allowed to accept multiple input and produce multiple output parameters. Additionally, the outputs produced by a component service should be allowed to be passed as inputs to the other components of the same composition.

Solution: As discussed in Section 2.1.3, each component service of a composition is represented as a function in an Indexical LUCID program and, therefore, in the LUCID segment of an OBJECTIVE LUCID program. In case a service generates multiple outputs, OBJECTIVE LUCID allows them to be composed into a JAVA object and returned from the service's definition in the JAVA segment to the LUCID segment. Individual output parameters (or data members) from these objects can then be accessed and passed as inputs/arguments to other service definitions/functions in the LUCID segment using the *dot* operator (as explained earlier in this section). For instance, consider the call to the *ReqComp* constructor in Listing 2.2 (line 107). Although it may appear to be superfluous in the given example where the *range* composite service is expected to produce only one output parameter, in case of composite services that produce multiple output parameters, outputs (potentially generated by different component services) can be passed as arguments to this constructor, assembled as data members of a JAVA object (such as *oRange*) and returned as the program output.

3 **Requirement:** A service should be allowed to execute only if all the constraints placed on it evaluate to *true*.

Solution: As discussed in Section 2.1.3, the *wvr* (or *whenever*) operator supplied by LUCID enables restrictions to be placed on expressions, including those that represent web services. Such expressions are computed, i.e., the services are executed, only if the conditions imposed on them are first evaluated to *true*. For instance, in order for the function *calcMax* in Listing 2.2 (line 119) to be computed, the conditions (line 126) that define its constraint variable *c_max* (line 120) must first evaluate to *true*.

4 **Requirement:** In a program representing a composition, simulated implementation of its component services should be allowed to be easily embedded as well as replaced, when required, with links to real services in order to facilitate an effortless transformation of service simulation into execution.

Solution: Declarative definitions of functions representing component services in an Indexical LUCID equivalent of a composite service are replaced in OBJECTIVE LUCID with procedural function definitions written in JAVA – a mainstream imperative language. Better familiarity with JAVA as compared to LUCID enables programmers to write placeholder implementation of component services and even reuse code that might already be available for simulation purposes much more conveniently. Moreover, swapping the simulation code of a service with an implementation that invokes the real online service itself can be easily accomplished in JAVA [38], thereby minimizing the effort involved in switching between simulation and execution of service compositions.

2.2 GIPSY

The *General Intensional Programming SYstem* (GIPSY) is a multi-language programming platform and demand-driven distributed execution environment for all LUCID dialects [24]. It is an ongoing project aimed at investigating the potential of the intensional programming model as realized by the latest versions of LUCID in varied domains. The architecture of GIPSY consists of three *tiers* or independent computational units – *Demand Generator Tier* (DGT), *Demand Worker Tier* (DWT) and *Demand Store Tier* (DST) – each of which is responsible for performing a specific set of tasks that form part of a program's execution process. In order for this process to be completed, all these tiers (whether deployed on the same or different computers) need to communicate and collaborate with each other, working as a group to achieve a common goal. The computers that register for hosting one or more of these tiers are known as *GIPSY nodes* whereas a set of interconnected GIPSY tiers deployed on these nodes executing GIPSY programs is known as a *GIPSY instance*. Communication between the tiers that together constitute a GIPSY instance is achieved through the generation, propagation and consumption of demands, thereby making the GIPSY Multi-Tier Architecture operational mode fully demand-driven. As discussed in Section 2.1.3, a demand in the eductive computation model is a request for the value of a program identifier in a specific context of evaluation. While the demands generated for the evaluation of LUCID identifiers are known as *intensional demands*, those generated for procedure identifiers, i.e., procedural function calls, embedded in a hybrid LUCID program are known as *procedural demands*.

Generation of demands, whether intensional or procedural, for a program begins at the Demand Generator Tier. This tier is responsible for traversing the abstract syntax tree (AST) representation generated by the GIPSY compiler, the *General Intensional Programming Compiler* (GIPC), for the declarative definition of each of the LUCID identifiers appearing in a program. For each of these identifiers, the DGT generates an intensional demand and sends it to the Demand Store Tier. The DST, also known as the *warehouse*, is tasked with persistently storing already computed demands along with their resulting values. It also acts as an asynchronous communication middleware between tiers in order to migrate demands and computed values between them. Upon receiving a new demand from a tier, the DST searches its records for the value of the demand in case it has already been computed. If found, the value is propagated back to the tier that requested it, otherwise the demand waits in the warehouse until a tier capable of computing it becomes available. An intensional demand whose value is not already available in the DST can be picked up for computation by the same DGT that generated it in the first place or by a different DGT, if available. Once computed, the resulting value of the demand is communicated to the warehouse so that it can be stored for future reference, thus achieving better processing performances by not having to re-compute the value of every demand every time it is eventually re-generated after having been processed. In case the demand was processed remotely by a DGT other than the one that generated it, the DST sends its computed value (after it has been recorded) to the original tier. As illustrated in Figure 6, the computation of a demand might depend on the values of other identifiers. In such a scenario, the DGT processing this demand generates further demands for these constituent identifiers and sends them to the DST for computation following the same process as followed for the evaluation of the original demand. Once computed, all these constituent values are communicated to this DGT, which then uses them to evaluate the original demand and returns the resulting value to the warehouse for storage.

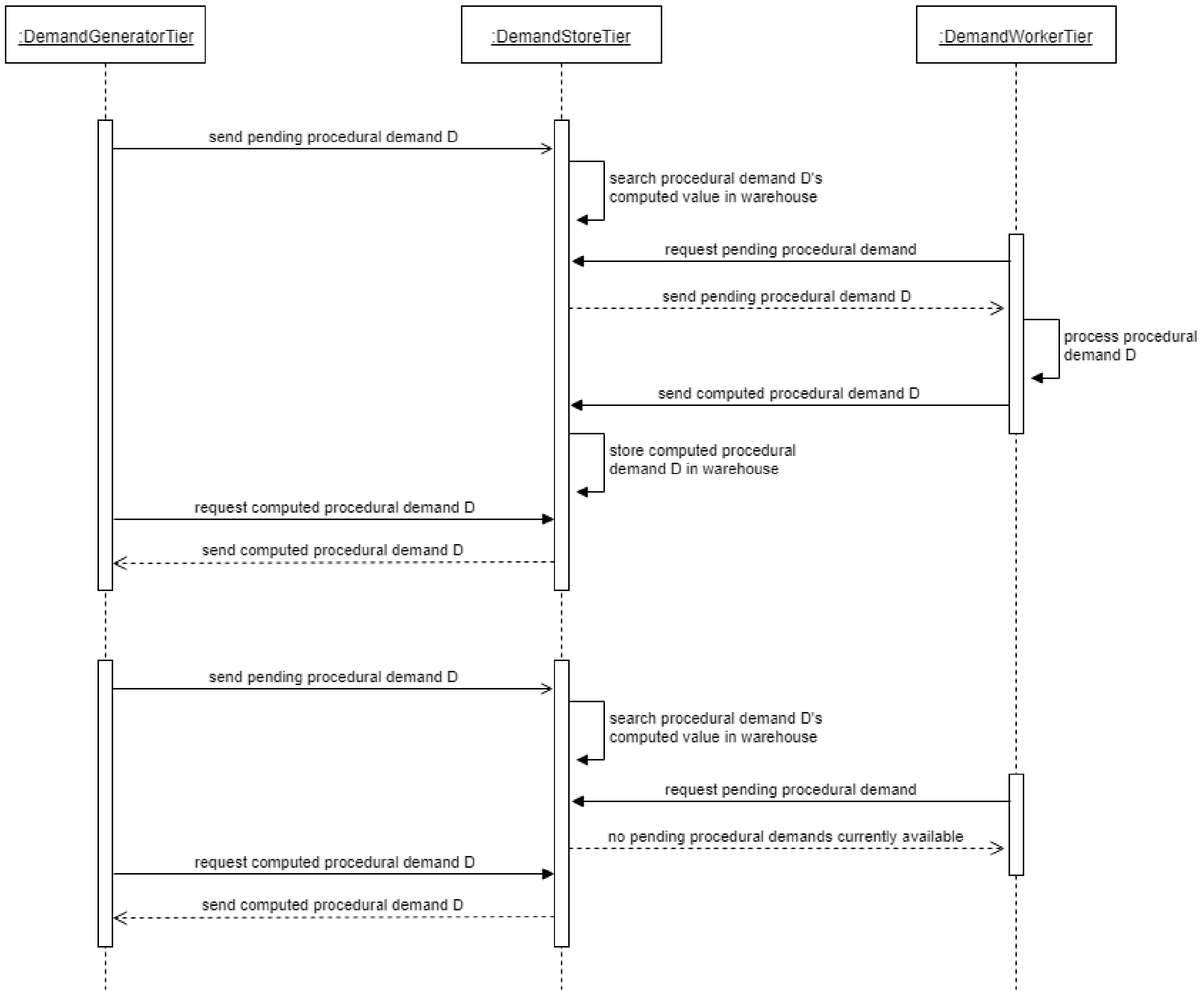

:DemandGeneratorTier
:DemandStoreTier
:DemandWorkerTier
send pending procedural demand D
search procedural demand D's computed value in warehouse
request pending procedural demand
send pending procedural demand D
process procedural demand D
send computed procedural demand D
store computed procedural demand D in warehouse
request computed procedural demand D
send computed procedural demand D
send pending procedural demand D
search procedural demand D's computed value in warehouse
request pending procedural demand
no pending procedural demands currently available
request computed procedural demand D
send computed procedural demand D

Figure 7: Processing of New and Previously-Computed Procedural Demand on GIPSY

A procedural demand is generated by a DGT when it encounters a procedural functional call node while traversing the AST of a Lucid identifier. Unlike intensional demands, a procedural demand can only be processed by a Demand Worker Tier, i.e., only a DWT can pick it for computation as it waits in the DST (in case its resulting value is not already available in the warehouse), execute the corresponding procedure written in an imperative language and send the resulting value back to the DST. The DST first stores this procedural demand and its computed value in the same way as intensional demands for future reference and then migrates the value back to the DGT that originally generated the demand. Figure 7 shows the sequence of actions taken by the DGT, DST and DWT for the computation of a new (i.e., not previously-computed) procedural demand D as well as the steps followed when the same demand is generated again and its resulting value is already available in the warehouse, thereby eliminating the need for re-computation.

2.3 Related Work

The primary goal of this thesis is to present a simulation/execution-based verification solution for context- and internal-constraint-aware composite web services. Therefore, our review of the existing literature on composite web services was focused on answering the following two questions:

Q1 Are there any studies that propose an approach for the verification of internal constraints imposed on composite web services? If there are, do they have any weaknesses that we overcome in our approach?

Q2 What kind of simulation/execution-based solutions exist, in general, for the verification and validation of composite web services? What similarities or differences do they present when compared to our solution?

In this section, we examine the findings of our review process that help us answer the above questions.

We begin the answer to **Q1** with a discussion on the research works conducted so far on composition of internally-constrained web services. In [3], Wang et al. acknowledge the fact that most web services can function correctly only within a specific context whose boundaries are defined by the constraints imposed on them by their providers, which we call *internal service constraints*, and that these constraints have a direct impact on the compositions that use them as components. They explain how a composite service can fail during execution if one or more constraints imposed on its component services are not satisfied despite all the input values being in perfect compliance with the input types required by the services. In order to avoid such failures, the authors propose a graph-search-based algorithm augmented with novel preprocessing techniques for constraint-aware composition of web services. Every component service in these compositions can potentially be replaced with a branched combination of services each of which can perform the same task but under different constraints (i.e. in different contexts). Which service from each group gets invoked at run-time depends on which service's constraints get completely satisfied in the given execution context. Although this approach, undoubtedly, widens the contextual range of a composite service, it does not guarantee exhaustive coverage of the context space, which is completely dependent on the combined contextual range of the services available for composition. Therefore, it is still possible for these composite services to fail during execution because of a constraint violation in case the input values do not fall within the combined contextual scope of their component service alternatives. Similar is the case with the research conducted by Laleh et al. [4, 5, 6, 7, 8]. The study proposes planning-graph-based algorithms for automated composition of internally-constrained web services driven by their input-output relationships. In order to reduce the number of component service rollbacks resulting from a constraint-verification failure during the execution of a composite service, the study offers a novel technique for adjusting the constraints in each composition plan to the earliest possible location at which they can be verified correctly. According to the study, once all constraint-aware composition plans are generated for a given composition request, they can be combined to form a larger package containing several alternative solutions for the same composition request. In case the internal constraints of one plan fail verification in a given execution context, it can be rolled back and the next plan in the package can be selected for execution. Although this approach serves to broaden the contextual scope of the composition solution while reducing the chance of failure due to constraint violation (similar to [3]), it does not completely eliminate the risk of run-time failures. Consequently, a verification system becomes necessary for detecting the scenarios or regions of execution context space in which a composite service could fail and for validating that it behaves as expected within its applicable contextual boundaries so that unexpected post-deployment failures along with their resulting damages (as discussed in Section 1.1) could be avoided or, at least, prepared for.

Using Petri nets for composition of internally-constrained web services allows for such a verification to a certain extent by means of simulation. For instance, in [30], Cheng, Liu, Zhou, Zeng, and Yla-Jaaski introduce an automatic composition method for internally-constrained fuzzy semantic web services using Fuzzy Predicate Petri Nets (FPPN's), where fuzzy semantics (or fuzziness) means syntactic and semantic representations involving fuzzy variables and fuzzy membership functions. In this method, a composition request is accepted from the user and its elements are modeled as a set of facts (user-provided inputs), rules (user-imposed behavioral constraints) and a goal statement (user-expected outputs) in the form of Horn clauses. Then, these Horn clauses are subjected to a T-invariant analysis technique to assemble a set of internally-constrained component services that can fulfill the user's fuzzy input/output and behavioral constraint requirements by ensuring that the services' internal constraints do not conflict with the requester constraints. The T-invariants are then modeled as an FPPN (a fuzzy extension to the standard predicate/transition nets) and analyzed to ensure complete absence of deadlocks in the composite service. Finally, reachability graph of Petri nets is used to determine the execution sequence for the components of the composite service. Depending on this sequence and the QoS parameter of each component service, the QoS value of the composite service can be calculated and used to select the optimal composition among all the ones generated for a given request. A similar approach towards constraint-aware web service composition is proposed by Zhu and Du in [12], which uses Logical Petri Nets (LPN's) – a high-level abstraction of Petri nets with inhibitor arcs – for modeling composite services. According to this approach, the input/output requirements obtained from a composition request are transformed into input parameters required by and output parameters produced by the services available for composition. Meanwhile, the user's behavioral and qualitative constraint requirements are formalized as logical expressions to guard the inputs and outputs of the resultant Petri net's transitions. During composition, those atomic services that not only satisfy the input/output requirements of the user but also exhibit internal behavioral constraints that match with the requester's constraints are selected while the others are rejected. Although the above two studies mainly focus on composition and not verification of services, it is common knowledge that Petri nets are capable of simulating the behavior of the systems that they represent and there are several tools available online that help users observe that simulated behavior [31, 32]. Therefore, it might be possible to use the above models for simulation-based verification of internal- and user-constraint-aware composite services. However, Petri nets do not have the capability to execute real services and, therefore, cannot be used for execution-based verification of composite services, which is essential to ensure that there are no discrepancies between a service's actual behavior and its description.

Another simulation-based verification technique for composite web services is introduced by Wang and Yu in [18]. As per this technique, the OWL-S process specification of the composite service to be verified is first translated into a finite state program written in the executable temporal logic language, called object-oriented MSVL, which is an executable subset of Projection Temporal Logic (PTL). The properties against which the service needs to be validated are expressed as formulas of Propositional Projection Temporal Logic (PPTL) – a specification language for describing desirable properties. The composite service program is then executed with the desirable property-formulas by the object-oriented MSVL interpreter to confirm if the service satisfies the properties. Being an object-oriented language, MSVL enables construction of better structured and understandable programs, thereby reducing potential errors. Moreover, it allows for a wide variety of composition constructs to be represented, including *Split*, *Join*, *Any-Order*, *If-Then-Else* and *Iterate*. However, this technique (same as Petri nets) lacks the ability to execute real services for analyzing their actual behavior. Moreover, other than stating that the components of a composite service may have certain pre-conditions placed on them, the authors do not discuss verification of internal service constraints, which leaves the relevance of this system to our research open to speculation.

A different approach towards constraint-driven composition of web services is proposed by Aggarwal, Verma, Miller, and Milnor in [9]. Using a composition framework known as METEOR-S (Managing End-To-End OpeRations for Semantic Web Services), this approach allows the components of an abstract process (composite service) to be bound to concrete web services selected based on business and process constraints, thereby generating an executable process. For creating the abstract process, this research uses BPEL4WS (Business Process Execution Language for Web Services) and augments the process activities with service templates, defining functional semantics and QoS specifications, that assist the constraint analyzer and execution engine modules of METEOR-S in matching concrete services to abstract placeholders. To enable METEOR-S to discover suitable concrete services for matching, this study proposes the use of semantically annotated WSDL service descriptions stored in an enhanced UDDI registry that has the METEOR-S discovery engine module as an interface. Selection of the optimal service from a set of potential candidates discovered for a process is based on their QoS specifications. Once the development, annotation, discovery and composition phases are complete, the resultant web process represented in BPEL4WS can be executed on the BPWS4J engine. Unlike the approaches discussed previously, this approach results in constraint-aware composite services that can be executed for testing actual behavior of real services. However, it first requires an abstract process to be constructed manually, which would grow increasingly impractical as the complexity of the composite service increases. Also, this solution can only support execution-based verification of composite services; it does not allow for simulation-based testing.

Table 4: Comparison of Research Works Concerning Internal Constraints (**Q1**)

Authors & Citations	**Year**	**Approach/Model/ Tool**	**Automated Composition**	**Internal Constraint Modeling**	**Simulation-based Verification**	**Execution-based Verification**
Aggarwal et al. [9]	2004	METEOR-S framework	+	+	−	+
Zhu and Du [12]	2010	Logical Petri Net	+	+	+	−
Wang et al. [3]	2014	Graph-search-based internal-constraint-aware composition	+	+	−	−
Cheng et al. [30]	2015	Fuzzy Predicate Petri Net	+	+	+	−
Wang and Yu [18]	2015	MSVL-PPTL verification	−	+	+	−
Laleh et al. [4, 5, 6, 7, 8]	2016-18	Planning-graph-based internal-constraint-aware composition	+	+	−	−

(+) Support, (-) No Support.

Based on the research works discussed so far, it can be concluded that the existing solutions for verification of internal constraints imposed on composite services rely either on simulation or execution but never support both. In contrast, the Lucid/GIPSY combination that we propose in this thesis has the capability of utilizing either technique with equal ease, which enhances the scope and, consequently, the reliability of the verification process. Additionally, part of our solution is an optimized version of the automated composition method defined by Laleh et al. [4, 5, 6, 7, 8], which incorporates a unique constraint-adjustment feature for reducing the rollback effort resulting from run-time constraint-verification failures, which, in turn, improves the overall efficiency of our simulation/execution process.

For answering **Q2**, we examine several of the simulation- and execution-based solutions proposed up to now for the validation of some or the other aspects of composite web services. Although these solutions are not concerned with verifying internal service constraints, drawing a contrast against them aids in effective examination of the strengths and limitations of our validation system. We begin this discussion with simulation-based approaches. Siala, Ait-Sadoune, and Ghedira [39] propose translating composite service descriptions written using WS-BPEL and WSDL into Multi-Agent Systems (MAS), which can simulate service behavior for observation and detection of undesired properties, such as, live-lock, deadlock, incorrect transformation of data and faulty ordering or termination of interactions, which, in turn, can be used for correcting the original composition descriptions, thereby enhancing the composite service's reliability. The authors define several rules for transforming WS-BPEL constructs into corresponding agent code and simulating them using the JADE (Java Agent DEvelopment) framework. However, their translation process itself is conducted manually, which would make it a cumbersome task for real-world compositions, which are usually large and complex in structure. Our translation framework, on the other hand, not only automates the transformation of composite services but also allows modular plugging-in and -out of modules for accommodating several different target models, including but not restricted to Lucid, XML and DOT [40, 41]. Additionally, parallel execution of activities in MAS is achieved through multi-threading, which demands a significant management effort of its own and is known to be a common source of complicated run-time errors. Lucid, however, being a dataflow programming langauge, is immune to such errors because of its inherently-concurrent nature, which eliminates the need for and, therefore, the risk involved in manual launching, synchronization and management of multiple threads (see Section 2.1.3).

Another similar approach to verification and validation of composite services is proposed by Narayanan and McIlraith in [16, 17]. They implement an interpreter for translating DAML-S composite processes into Petri nets that can be simulated on the KarmaSIM simulation and modeling environment. The Petri net based KarmaSIM tool allows them to achieve interactive and visual simulation of service compositions, model component service concurrencies and check for reachability, liveness and existence of deadlocks in composite services. A clear advantage of this system over ours is its graphical representation of the service networks being simulated, which aids in enhanced visualization, particularly in case of unfamiliar or complex processes. In order to compensate for the lack of a graphical interface in our simulation tool, we exploit the capabilities of our modular translation framework and implement a module for generating DOT graph representations of layered composite services, which enables them to be visualized graphically.

The use of Petri nets for simulation and analysis of composite web services is also proposed by Juan and Hao in [33]. Their approach makes use of QPME - a performance modeling and analysis tool based on Queuing Petri Net (QPN) modeling formalism. According to this approach, the QPN Editor (QPE) component of QPME is first used to transform WS-BPEL processes into QPN's. Then, the SimQPN (Simulator for QPN's) component of the tool is used to simulate the QPN's so constructed for quantitative performance analysis. Results of this analysis as presented in [33] reveal that the performance of web service compositions can be improved by adjusting the queuing parameter values. Yet another variant of Petri nets, known as Colored Petri Net (CPN), is suggested for formal modeling of WS-BPEL processes by Dechsupa, Vatanawood, and Thongtak in [19]. This study defines a set of simple rules for transforming WS-BPEL descriptions of composition processes into Colored Petri nets – classical Petri nets augmented with data, hierarchy and time. It then uses the CPN Tool for editing, simulation and analysis of the transformed CPN's. Unlike many other Petri net-based solutions, this system not only checks for deadlocks and reachability in composite services but also validates their behavior in response to valid and invalid input values. To perform such behavioral tests, this approach generates dummy definitions/stubs for each component service, which are then used in conjunction with the equivalence class partitioning technique for preparing equivalence classes of service input/output parameters and determining their range of values – both valid and invalid. The drawback, however, is that this approach does not allow more than two parameters per service operation, which places a major restriction on the variety of compositions that can be validated using this system. Meanwhile, no such restriction exists in the verification solution that we propose.

Another analysis technique involving translation of WS-BPEL processes into a verification language is discussed by Fu, Bultan, and Su in [20] and by Nagamouttou, Egambaram, Krishnan, and Narasingam in [42]. Both of these research works propose translation of composite services written using WS-BPEL into an automata model serving as an intermediate representation, which, in turn, is translated into the Promela language. The Simple ProMeLa Interpreter (SPIN) tool is then used for analyzing these Promela translations of service compositions for various desirable and undesirable properties. The system presented by Fu et al. is mainly concerned with analyzing and synchronizing the asynchronous XML messages exchanged among component services to ensure that the services get executed in the correct order. This system uses guarded automata augmented with unbounded queues for incoming messages as the intermediate representation for its modular and extensible translation framework (similar to ours), which can support multiple web service specification languages at the front-end and various model checking tools at the back-end. On the other hand, Nagamouttou et al. are more concerned with checking their Promela translations for deadlocks, dead transitions, reachability and liveness properties. This approach first collects the user request for composition, uses it for invoking the related atomic services, requires them to be composed into a WS-BPEL process manually (as opposed to our automated composition methodology) and then feeds the composed process as input to the verification unit where it is translated into the Enhanced Stacked Automata Model (ESAM), which is the intermediate representation of this system.

A different approach to analyzing composite services described using WS-BPEL is presented by Chen, Tan, Sun, Liu, and Dong in [13]. Instead of translating a WS-BPEL process into an intermediate modeling language for verification, which, according to them opens the system to concurrency bugs that accompany multi-threading techniques used to represent parallel activities, the authors present a tool called VeriWS that can directly analyze the semantics of a WS-BPEL process to check for deadlock-freeness, reachability and QoS constraint-satisfaction. Additionally, the tool also provides a simulator that helps observe the behavior of a WS-BPEL composition, which, in turn, helps detect anomalies in its functionality. The simulator also aids in visualizing the WS-BPEL counterexample that gets generated by the verifier component of the tool in case any non-functional requirement violations are detected. A significant similarity between this solution and ours is their modular and extensible software architectures that allow new verifiers to be plugged-in to VeriWS and new translators to our translation framework, thereby making them more flexible and versatile. Also, it should be noted that the concurrency bugs that this system avoids by omitting translation to a formal language pose no threat to our system either, owing to the inherent concurrency of LUCID's dataflow execution model, which eliminates the need for manual multi-threading altogether (see Section 2.1.3).

A two-stage process for validating composite web services described using WS-BPEL is proposed by Shkarupylo in [15]. The first step of the process is to synthesize formal TLA+ (Temporal Logic of Actions) specification for a composite service, defining the functional properties required of the service. Once formulated, this TLA+ specification is used by the TLC model checker for testing if the WS-BPEL description of its corresponding service exhibits all the required functional properties. If the WS-BPEL description is found to be consistent with the requirement specification at this stage, DEVS (Discrete Event System Specification) simulation models for the composite and component services are designed based on the TLA+ specification. These models are then provided to the DEVS Suite toolkit to enable simulation and visualization of the service behavior and validation of its functional and non-functional properties.

Adadi, Berrada, Chenouni, and Bounabat [43] propose translation of a service composition constructed using a multi-agent system into a WS-BPEL process for simulation, which is the inverse of the approach presented by Siala et al. in [39] (discussed earlier in this section). In this approach, a special type of multi-agent system, known as MARDS (Multi-Agent Reactive Decisional System), is used for composing web services each of which is represented by a DRA (Decisional Reactive Agent). Although ultimately intended for use as a verification system for composite services, this solution has the drawback of not being mature enough (by the time [43] was published) to be able to use its simulation technique for verification purposes. Same is the case with the end-to-end response-time analysis system proposed by Youcef, Bhatti, Mokdad, and Monfort [11] for synchronous and asynchronous composite web services invoked over the Internet. Although this system successfully uses a discrete event queuing network model for simulating the execution time of web services, an emulator capable of validating the results of that simulation is still part of its future work.

Having discussed several verification systems that are purely based on simulation, we now examine the Triana framework, which relies on execution for validating composite services. As described by Majithia, Shields, Taylor, and Wang in [44], Triana is an open source, distributed, platform-independent Problem Solving Environment (PSE) written in Java that allows users to discover web services either by querying UDDI registries or by specifying their WSDL locations, compose the discovered services into workflows through a graphical interface, export the composed services as a BPEL4WS graph or as a service that can be registered with UDDI, execute the compositions for analysis by invoking component services using SOAP over HTTP and, if required, alter the compositions by re-arranging their workflows on the canvas. When compared to our proposed solution, Triana exhibits several similarities such as support for composing web services, ability to export composed services for reuse and an extensible framework that enables easy inclusion of additional export formats. On the other hand, Triana's graphical interface makes it more user-friendly in comparison to our system, which we compensate for to some extent through our DOT graph translator module. Meanwhile, the factor that provides an edge to our system over Triana is its ability to compose services automatically as opposed to Triana's manual discovery and composition process, which would prove to be an unwieldy task while handling large repositories and complex compositions.

Moving on to solutions that can exploit the capabilities of both simulation and execution techniques for verification purposes, we begin by examining the CRESS (Chisel Representation Employing Systematic Specification) representation and toolset proposed for automated translation and analysis of composite services by Turner [45]. This toolset transforms a composite service diagram drawn using a graphical editor into a CRESS directed graph, which, in turn, is translated into the target language whose framework details are also provided as input to it. To facilitate automated analysis, CRESS supports translation of composite services to formal specification languages, such as, LOTOS and SDL (which can be validated on their specific simulation-based analyzers), whereas to facilitate service execution, it supports their translation to BPEL and WSDL (which can be deployed using ActiveBPEL and other similar BPEL environments). Verification checks supported for

CRESS translations include those for deadlocks, live-locks, liveness and consistency of high level descriptions with their detailed designs. In addition to these, use-case scenarios can also be defined and translated into target language test processes using CRESS, which can then validate the translated composite service's behavior in specific usage situations. MUSTARD is one of the independent yet related tools that can be used for defining use-case scenarios for such validation checks. Features shared by CRESS and our solution include their support for graphical representation of composite services to aid visualization, automated and extensible translation framework, ability to simulate as well as execute composite services and facility to manually implement component services in case their actual definitions are unavailable. A slight drawback of CRESS is the effort it demands for representing parallel execution of services, which is not required in our solution as it is handled automatically in Lucid.

Yet another system that can use both execution and simulation techniques for analysis of composite web services is the JSIM-SCET tool set proposed by Chandrasekaran et al. and Silver et al. in [46, 47, 48, 49]. The Service Composition and Execution Tool (SCET) provides a graphical designer, known as Web Process Design Tool (WPDT), which allows service compositions to be visualized and manually constructed (as opposed to our automated composition technique) as digraphs. Descriptions of the web processes so constructed can be stored as Web Service Flow Language (WSFL) based specifications within the designer or as XML documents in a repository. In order to facilitate execution of the composed services, their WSFL specifications can be automatically transformed into Perl code by the Perl Execution Code Generator sub-module of SCET. Execution of this code for functionality and performance analysis is handled by the Perl execution controller module. Alternatively, the WSFL specifications can be used by the Simulation Model Generator unit of SCET for generating Java-based specifications that can be simulated on the JSIM simulator. Simulation is used as an alternative to execution in this system when the component services being analyzed are either world-altering or involve an invocation cost, which makes their execution impractical and expensive. Based on the results of the analysis (whether simulated or executed), the compositions being analyzed can be tuned so as to improve their efficiency

Table 5: Comparison of Simulation/Execution-based Verification Approaches (Q2)

Authors & Citations	**Year**	**Approach/Model/ Tool**	**Automated Composition**	**Internal Constraint Modeling**	**Simulation-based Verification**	**Execution-based Verification**
Chandrasekaran et al. & Silver et al. [46, 47, 48, 49]	2002-03	JSIM-SCET tool set	−	−	+	+
Narayanan and McIlraith [16, 17]	2002-03	DAML-S to Petri Net translation	+	−	+	−
Fu et al. & Nagamouttou et al. [20, 42]	2004 & 2015	WS-BPEL to Promela translation	−	−	+	−
Majithia et al. [44]	2004	Triana Problem Solving Environment	−	−	−	+
Youcef et al. [11]	2006	Discrete event queuing network	−	−	−	−
Turner [45]	2007	CRESS representation & tool set	−	−	+	+
Juan and Hao [33]	2012	WS-BPEL to Queuing Petri Net translation	−	−	+	−
Chen et al. [13]	2014	WS-BPEL analysis on VeriWS	−	−	+	−
Siala et al. [39]	2014	WS-BPEL to Multi-Agent System translation	−	−	+	−
Adadi et al. [43]	2015	Multi-Agent System to WS-BPEL translation	−	−	−	−
Dechsupa et al. [19]	2016	WS-BPEL to Colored Petri Net translation	−	−	+	−
Shkarupylo [15]	2016	TLA- & DEVS-based verification of WS-BPEL	−	−	+	−

(+) Support, (-) No Support.

measured in terms of their execution time. Features that this system has in common with our proposed solution include its graphical representation of web processes, ability to store them after composition for further processing, automated translation methodology and support for simulation as well as execution of service compositions.

Based on the research works discussed in response to **Q2**, it can be concluded that none of these systems other than CRESS and JSIM-SCET have the capability to employ both simulation and execution for composite service verification. Most of these systems use WS-BPEL processes as their input, which they first translate into a verification language or a formal simulation/execution model and then validate using a simulator, execution controller or model checker tool. Although several of these approaches succeed in incorporating automation and extensibility features in their translation processes, none of them supports completely automated composition of services, which limits their practicality while handling complex compositions or large repositories of component services. Even widely-used open-source and commercial web service testing tools, such as SoapUI [50], while exhibiting simulation and execution capabilities, fail to support examination of internally-constrained and composite web services. In contrast, the solution that we propose in this thesis not only enables automated composition of services but also includes a unique constraint-adjustment feature (introduced by Laleh et al. [4, 5, 6, 7, 8]) for improving the efficiency of the process. Additionally, the inherent concurrency of LUCID's dataflow execution model, unlike most of the other systems, exempts our approach from the effort and risk involved in handling parallel execution of services and multiple thread management. Another factor that differentiates other existing systems from ours is their purpose of verification. While most of these studies concern themselves with checking for component-ordering, deadlock, live-lock, reachability and QoS properties, we are focused on validating internal constraints placed on services by their providers. Based on the above discussion, the factors that we found our system to be lacking in when compared to other existing systems include an interactive graphical interface for service composition and verification and the ability to represent *if-else* and looping constructs in composite services. In order to compensate for the lack of a graphical interface and aid visualization, we incorporate a module in our translator for generating DOT graph representations of service compositions. However, being based on Laleh's planning-graph approach to service composition, it is not possible for our system to model uncertainties or repetition in execution of components (as mentioned in Section 1.4). The only conditions that are allowed during execution, at present, are those resulting from internal service constraints. In the future, however, we plan to extend our current system to merge all compositions generated for a request into a single package (as proposed by Laleh et al.), containing alternative sub-plans to choose from based on runtime constraint-satisfaction in order to help recover from execution failures (see Section 6.2).

2.4 Summary

The inherent concurrency of LUCID's dataflow execution model, the ability of GIPSY to store execution results with respect to their context in its warehouse for future reference and the ability of the LUCID/GIPSY system to both simulate and execute composite services with equal ease, together with an automated composition methodology with its unique constraint-adjustment technique (designed by Laleh et al.) and an extensible and modular translation framework, are all characteristics that distinguish the system presented in this thesis from other composite service verification solutions. In this chapter, we examined such distinctive features of LUCID and GIPSY in detail and compared our proposed solution with other related research works conducted in the field to this date to gain a clear understanding of its strengths and limitations. With this understanding as the base, from the next chapter onwards, we start exploring our research methodology (outlined in Section 1.5) in greater detail, beginning with a discussion on Laleh's unique automated composition technique, the steps that we take to complete and optimize it as well as its reimplementation as a more flexible, modular and maintainable application.

Chapter 3

Service Composition

As mentioned in Contribution **1** (Section 1.3), while simulating and executing composite web services with internal constraints is the primary goal of this thesis, 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 (see Section 1.5). The automated service composition technique that we employ in this thesis has been borrowed from the research conducted by Laleh et al. [4, 5, 6, 7, 8]. In this chapter, we discuss this unique composition methodology, the layered structure of the composite services that it generates, the additions and modifications that we make to complete and optimize this technique and the specific features that we introduce during its re-implementation to transform it into an independent, flexible and maintainable application.

3.1 Composite Service Model

In order to understand the service composition methodology devised by Laleh et al., it is essential to have a clear understanding of the fundamental entities and concepts involved in it. In this section, we present the formal definitions of such entities and concepts as provided by Laleh in [4].

Definition 1. *A **Service** is a tuple* $S = \langle I, O, C, E, QoS \rangle$ *where:*

- ***I*** *is the set of ontology types representing the input parameters of the service.*
- ***O*** *is the set of ontology types representing the output parameters of the service.*
- ***C*** *is the set of constraint expressions representing limitations on service features.*
- ***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.*

For instance, elements of the Payment service W_3 listed in Table 6 would be expressed as:

- $I = \{OrderNumber, PaymentAmount, CreditCardBrand, CreditCardNumber\}$
- $O = \{PaymentStatus\}$
- $C = \{CreditCardBrand = Visa\}$
- $E = \{PaymentStatus\}$
- $QoS = \{\}$

In this thesis, we do not take QoS features or their related constraints into consideration. Therefore, although we incorporate a placeholder for QoS parameters in our implementation of the Service entity, its sole purpose is to complete the Service structure, which would be required for verification of QoS constraints in the future. At present, we focus only on the constraints imposed on services by their providers, which are known as *internal constraints*. Although we have already defined these constraints in Section 1.1, here, we present a more formal definition for them derived from the generic definition of constraints provided by Laleh.

Definition 2. *An **Internal Constraint** is an expression that can be evaluated to either true or false. For simplicity, we restrict ourselves to expressions of the form:*
$\langle feature\rangle\ \langle operator\rangle\ \langle literalValue\rangle$*, 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.*

For instance, the internal constraint placed on the CreditCardBrand input parameter of Payment service W_3 listed in Table 6 would be expressed as $CreditCardBrand = Visa$.

As explained in the research methodology section (1.5), in order to initiate automated composition of services, besides a set of services available to be used as components, a composition request, specifying the user's requirements, is also required. This composition request is defined as follows:

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.*

For instance, elements of the request for composing the services listed in Table 6 to construct an online shopping service similar to the one depicted in Figure 2 would be expressed as:

- $I = \{ProductName, CreditCardBrand, CreditCardNumber, ShippingAddress\}$

- $O = \{ShipmentStatus\}$

- $QoS = \{\}$

- $C = \{\}$

Since, in this thesis, 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. At present, they are included only for the sake of completeness of the composition model. In response to a composition request, Laleh's composition approach generates 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 their components. These plans are termed as constraint-aware plans and are defined as:

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 is eventually generating the goal parameters (R.O).*

The search graph referred to in the above definition is the graph of search 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 the given composition request (explained further in Section 3.2, depicted in Figure 9). 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.*

For instance, predecessor and successor sets for Shipment service W_7 depicted in Figure 13 would be: $predecessors(W_7) = \{W_1, W_4\}$ and $successors(W_7) = \emptyset$.

3.2 Service Composition Algorithms

Laleh et al. present their 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. However, while they completely design the other algorithms (*ServiceComposition*, *ForwardExpansion*, *AddService*, *BackwardSearch* and *AdjustConstraint*) participating in the process, only a brief description is provided for the *ConstructPlan* algorithm referred to in line 7 of the *ServiceComposition* algorithm (1). Since, the *ConstructPlan* algorithm is an integral part of and is essential for completing the implementation of the service composition process, we design it (Algorithm 5) as part of this thesis based on its description provided in [5]. In this section, we present and explain this algorithm along with the others (as presented in [5]) that together constitute the planning-based constraint-aware composition methodology (summarized in Figure 8).

Algorithm 1 drives the service composition process, invoking the other algorithms as and when required. It consists of four major steps: (1) *forward expansion* (Algorithm 2) responsible for constructing a *search graph* based on the given composition request and available services (line 2), (2) *backward search* (Algorithm 4) for extracting *solution plan sets* from the search graph (line 6), (3) *plan construction* (Algorithm 5) tasked with discarding extraneous services from the solution plan sets and arranging the remaining ones into workflows or *solution plans* (line 7) and (4) *constraint-aware plan construction* for transforming the solution plans into *constraint-aware plans* (lines 13 - 18) with their constraint verification points adjusted to optimum locations (Algorithm 6). The algorithm results in failure if no constraint-aware plans can be generated for the given composition request.

Algorithm 2 is responsible for generating a search graph for the given composition request (R). A search graph is a directed graph composed of ordered layers, each of which is assigned certain specific services selected from the given repository (SR). Assignment of services to layers depends upon their input-output relationship with each other. The

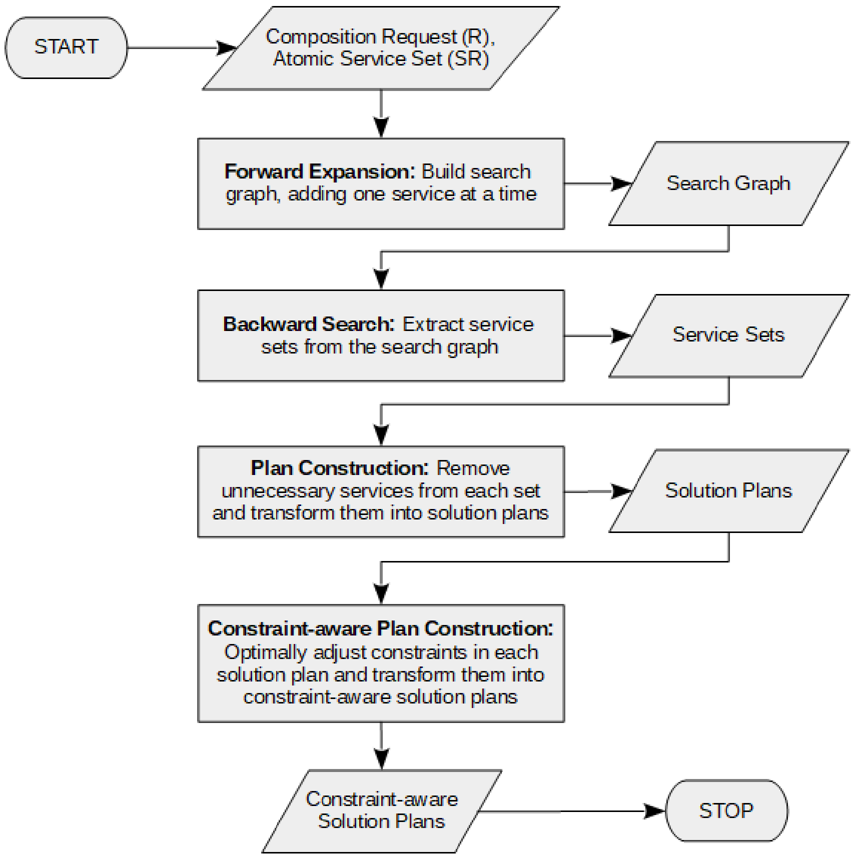

START
Composition Request (R), Atomic Service Set (SR)
Forward Expansion: Build search graph, adding one service at a time
Search Graph
Backward Search: Extract service sets from the search graph
Service Sets
Plan Construction: Remove unnecessary services from each set and transform them into solution plans
Solution Plans
Constraint-aware Plan Construction: Optimally adjust constraints in each solution plan and transform them into constraint-aware solution plans
Constraint-aware Solution Plans
STOP

Figure 8: Planning-based Constraint-aware Service Composition Methodology

services that generate output parameters that serve as inputs to other services are placed in earlier layers while the services that consume their outputs are placed in later ones. In order to find services that fulfill the user's requirements and determine the relationship between them, $prdSet$ (i.e. the predecessor set) is used. Its job is to keep track of the parameters produced by each successive layer of services as it gets added to the search graph. As forward expansion begins, $prdSet$ is initialized with the input parameters of the composition request ($R.I$), following which the repository is searched for services all of whose input parameters can be found in $prdSet$. Each of the suitable services discovered during the search must be able to generate an output parameter that is not already included in $prdSet$ (lines 2 - 5) and must not violate any of the constraints ($R.C$) specified in the composition request (line 7). Services that match all these criteria are added to the next layer in the search

Algorithm 1 ServiceComposition

```
Input: $R$ (composition request), $SR$ (set of available services).
Output: $plans$ (a set of constraint-aware plans, or failure).
 1: $serviceSet = \emptyset; plans = \emptyset$
 2: $searchGraph = ForwardExpansion(R, SR)$
 3: repeat
 4:    $l$ = maximum layer index in the search graph
 5:    $ServiceSet$ = all services in layer l of the search graph
 6:    $serviceSet = BackwardSearch(searchGraph, ServiceSet, \emptyset, l)$
 7:    $plan = constructPlan(serviceSet)$
 8:    if ($plan \notin plans$) then
 9:       $plans = plans \cup plan$
10:    end if
11: until ($No\ more\ plan\ can\ be\ added\ to\ the\ plans$)
12: if ($plans \neq \emptyset$) then
13:    for (each $plan \in plans$) do
14:       for (each $service \in plan$) do
15:          $serviceNode.service = service$
16:          $serviceNode.C_s = service.C$
17:          $cnstrAwarePlan = cnstrAwarePlan \cup serviceNode$
18:       end for
19:       $cnstrAwarePlan = adjustConstraint(cnstrAwarePlan)$
20:       $cnstr\_plans = cnstr\_plans \cup cnstrAwarePlan$
21:    end for
22:    return $cnstr\_plans$
23: else
24:    return $failure$
25: end if
```

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. In this way, the search graph grows layer by layer until no more services from the repository can be added to it. If the *prdSet* obtained at the end of the expansion is found to contain all the requested output parameters ($R.O$), the problem is considered solvable and the search graph is returned to Algorithm 1 for further processing (lines 15 - 17), otherwise, the problem cannot be solved by the given set of services and forward expansion ends in failure.

Algorithm 3 is invoked by the forward expansion process for determining the location

Algorithm 2 ForwardExpansion

```
Input: $R$ (composition request), $SR$ (set of available services)
Output: $searchGraph$ (search graph generated by forward expansion).
1: $searchGraph = null; prdSet = R.I$
2: repeat
3:    $layerSet = \emptyset$
4:    for each $service$ in $SR$ do
5:       if ($service.I \subseteq prdSet$) and ($service.O - prdSet \neq \emptyset$) then
6:          $l = AddService(searchGraph, service)$
7:          if ($CheckRequesterConstraints(l, R.C)$) then
8:             $searchGraph = l$
9:             $layerSet = layerSet \cup service.O$
10:         end if
11:      end if
12:   end for
13:   $prdSet = prdSet \cup layerSet$
14: until ($No\ service\ could\ be\ added\ to\ the\ search\ graph$)
15: if ($R.O \subset prdSet$) then
16:    return $searchGraph$
17: end if
18: return $failure$
```

at which a newly selected service (*newService*) should be inserted in the search graph. To accomplish that, Algorithm 3 scans the search graph constructed until the discovery of *newService* from the first to the last layer (*maximum layer index*) sequentially and finds the services one or more of whose output parameters serve as inputs to *newService*. All such services are added to the predecessor set of *newService* (lines 4 - 8). Once all predecessors are found, *newService* is inserted into the layer after the one that contains its latest predecessor. This approach allows services to be composed in sequence or in parallel.

Algorithm 4 recursively extracts a sequence of service sets from the search graph using a backward-chaining strategy, which can reach the goal parameters ($R.O$) from the initial parameters ($R.I$). Each service set in this sequence belongs to a different layer of the graph. Starting from the last layer, each time the algorithm backtracks, it selects a subset of services (*serviceSet*) from the power set of predecessor services (*preSrvSet*) from the previous layer of the services selected in the last recursion (lines 1 - 4). In case the selected subset does

Algorithm 3 AddService

```
Input: $searchGraph$ (a search graph), $newService$ (A new service)
Output: $searchGraph$ (a search graph includes $newService$)
1: $lyr = 0; newIn = newService.I$
2: while ($lyr \leq$ maximum layer index in $searchGraph$) do
3:    $serviceLayerSet$ = all services in layer $lyr$ of searchGraph
4:    for (each $service \in serviceLayerSet$) do
5:       $prdSet = service.O \cap newIn$
6:       if ($prdSet \neq \emptyset$) then
7:          $newService.predecessor = newService.predecessor \cup service$
8:          $newIn = newIn - prdSet$
9:          $Searchgprah.serviceSet = Searchgprah.serviceSet \cup newService$
10:      else
11:         return $searchGraph$
12:      end if
13:   end for
14:   $lyr = lyr + 1$
15: end while
```

not include any of the predecessor services (lines 5 - 7), it is ignored and the algorithm continues with another subset from the same predecessor power set. If the backtracking reaches the first layer of the search graph, the set of input parameters of all the services selected from the first layer is inspected to ensure that they all are available in the set of initial parameters (lines 10 - 15), otherwise, this *planSet* is ignored and backward search continues with another subset from the first layer (lines 13 - 15). If the input parameters of initial services are found to be satisfactory, the set of output parameters of all the services belonging to this *planSet* is checked to ensure that it includes all the goal parameters (lines 16 - 21). If successfully verified, the *planSet* is returned to Algorithm 1 to be transformed into a solution plan (*plan*).

The function *constructPlan* (Algorithm 1, line 7) has been described in [5] as being responsible for discarding all the unnecessary services from the *serviceSet* (or *planSet* from Algorithm 4) in order to minimize the number of component services in the final solution and then arranging the remaining services in the *serviceSet* in proper sequence to form a solution plan. Based on this description, we have designed **Algorithm 5** to first organize the services

Algorithm 4 BackwardSearch

Input: *searchGraph* (a search graph on which the BackwardSearch is applied), *preSrvSet* (set of predecessor services), *planSet* (the set of services in the solution plan), l (the layer number from which to start the search)

Output: *planSet* or failure

```
1:  $S$ = all services in layer $l$ of the search graph
2:  $serviceSet = preSrvSet \cap S$
3:  $planPowerSet = powerSet(S)$
4:  for (each $set \in planPowerSet$) do
5:      if (($serviceSet \cap set) = \emptyset$) then
6:          Continue
7:      end if
8:      $planSet = planSet \cup set$
9:      if ($l = 1$) then
10:         for (each $service \in set$) do
11:             $inputSet = inputSet \cup service.I$
12:         end for
13:         if ($inputSet \not\subset R.I$) then
14:             $Continue$
15:         end if
16:         for (each $service \in planSet$) do
17:             $outputSet = outputSet \cup service.O$
18:         end for
19:         if ($R.O \subset outputSet$) then
20:             return $planSet$
21:         end if
22:     else
23:         for (each $service \in set$) do
24:             $preSet = preSet \cup service.predecessors$
25:         end for
26:         if ($preSet \neq \emptyset$) then
27:             return $BackwardSearch(searchGraph, preSrvSet \cup preSet, planSet, l - 1)$
28:         else
29:             return $Failure$
30:         end if
31:     end if
32: end for
```

Algorithm 5 ConstructPlan

Input: R (composition request), $serviceSet$ (a set of services generated by BackwardSearch)
Output: $plan$ (a solution plan)

```
1: $m$ = maximum layer index of services in $serviceSet$
2: for ($l = 0$ to $m$) do
3:    $lIndexedSvcSet$ = all services in $serviceSet$ with layer index $l$
4:    $plan = plan \cup lIndexedSvcSet$
5: end for
6: repeat
7:    for ($l = 0$ to $m - 1$) do
8:       $layerSet$ = all services in layer $l$ of $plan$
9:       for (each $service \in layerSet$) do
10:         for (each $predSvc \in service.predecessors$) do
11:            if ($predSvc \in plan$) then
12:               $predOutpSet = predOutpSet \cup predSvc.O$
13:            end if
14:         end for
15:         if ($service.I \not\subset predOutpSet \cup R.I$) then
16:            $plan = plan - service$
17:         end if
18:      end for
19:   end for
20:   for ($l = m - 1$ to $0$) do
21:      $layerSet$ = all services in layer $l$ of $plan$
22:      for (each $service \in layerSet$) do
23:         if ($service.successors \cap plan = \emptyset$) then
24:            if ($service.O \cap R.O = \emptyset$) then
25:               $plan = plan - service$
26:            end if
27:         end if
28:      end for
29:   end for
30: until ($No\ more\ services\ can\ be\ removed\ from\ plan$)
31: if ($|plan| > 1$) then
32:    for (each $service \in plan$) do
33:       $planOutpSet = planOutpSet \cup service.O$
34:    end for
35:    if ($R.O \not\subset planOutpSet$) then
36:       return $\emptyset$
37:    else
38:       return $plan$
39:    end if
40: else
41:    return $\emptyset$
42: end if
```

in $serviceSet$ in order of their layer indexes (lines 1 - 5), resulting in the creation of a layered directed graph ($plan$), which can be viewed as a part of the original search graph. Then, for each service in the $plan$ (iterating from first to last layer), the set of output parameters of all its predecessors that are included in the $plan$ together with the initial parameters ($R.I$) is checked to ensure that it includes all the input parameters of the service, otherwise, the service is removed from the $plan$ (lines 7 - 19). Each of the remaining services in the $plan$ (iterating from last to first layer) that neither have a successor in the $plan$ nor produce any of the goal parameters ($R.O$) as output are also discarded (lines 20 - 29). This verification and removal process continues until no more services can be removed from the $plan$. For the resultant $plan$ to qualify as a composition of services, it must contain at least two services (line 31). Additionally, the set of output parameters of all its remaining services must include all the goal parameters (lines 32 - 35). If the $plan$ satisfies these conditions, it is returned to Algorithm 1 as a solution plan (line 38), otherwise, it is discarded as invalid (lines 36 and 41).

Algorithm 6 adjusts the constraint verification points within a constraint-aware plan (generated by Algorithm 1, lines 13 - 18) to optimal locations. From the second layer onwards, each constraint of each service-node ($serviceNode$) in $constraintPlan$ is moved to as early a layer as feasible (lines 9 - 28). To accomplish that, a service-node ($preNode$) that belongs to the set of predecessors ($preSet$) of $serviceNode$ and affects the value of the feature to which the constraint applies is selected (lines 15 - 16). The constraint is then moved to verification points immediately before the execution of all successor service-nodes of $preNode$ (lines 17 - 19). 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 $constraintPlan$. In case no predecessors are found to affect the constrained feature's value, the constraint is moved to the beginning of $constraintPlan$ (lines 11 - 13). Once all the constraints in $constraintPlan$ are similarly adjusted to optimal verification points, it is returned to Algorithm 1 as one of the constraint-aware solutions to the given composition problem.

Algorithm 6 AdjustConstraint

```
Input: $constraintPlan$ (a constraint-aware plan)
Output: $constraintPlan$ (a constraint-aware plan with adjusted constraints)
1: $l_1 = 2$
2: while ($l_1 \leq$ maximum layer index in $constraintPlan$) do
3:   $layerSet$ = all service-nodes in layer $l_1$ of $constraintPlan$
4:   $l_2 = l_1 - 1$
5:   $preLayerSet$ = all service-nodes in layer $l_2$ of $constraintPlan$
6:   for (each $serviceNode \in layerSet$) do
7:     $preSet = serviceNode.predecessors \cap preLayerSet$
8:     $constraintSet = serviceNode.service.C$
9:     for (each $constraint \in constraintSet$) do
10:      repeat
11:        if ($preSet = \emptyset$) then
12:          Add the constraint to the beginning of the $constraintPlan$
13:          $break$
14:        end if
15:        preNode = a node of preSet with the highest layer
16:        if ( $constraint.feature \in preNode.service.E$) then
17:          for (each $sNode \in serviceNode.successors$) do
18:            $sNode.C_s = sNode.C_s \cup constraint$
19:          end for
20:          $constraintSet = constraintSet - constraint$
21:          $break$
22:        else
23:          $preSet = preSet - preNode$
24:          $preSet = preSet \cup preNode.predecessors$
25:        end if
26:      until ($preSet \neq \emptyset$)
27:    end for
28:  end for
29:  $l_1 = l_1 + 1$
30: end while
31: return $constraintPlan$
```

3.3 Service Composition Example

In this section, we present an example to demonstrate the step-by-step construction of constraint-aware plans for a given composition request and repository of available services. Consider a composition request (R) for constructing an online shopping service similar to the one depicted in Figure 2, such that:

- $R.I = \{ProductName, CreditCardBrand, CreditCardNumber, ShippingAddress\}$
- $R.O = \{ShipmentStatus\}$
- $R.QoS = \{\}$
- $R.C = \{\}$

Details of the services available for the composition (i.e., the repository SR) are provided in Table 6 – an extended version of Table 1. Based on these R and SR, the forward expansion stage of the composition process constructs a search graph as depicted in Figure 9. The *prdSet* is initialized with the initial parameters ($R.I$), gets incrementally augmented with the new parameters produced by services of each new layer added to the graph and, finally, reaches the state where it contains all the goal parameters ($R.O$). The services that get added to each layer of the graph and the inputs they consume and outputs they produce are also shown in Figure 9. Shipment service W_8 cannot be included in this search graph because all its required input parameters get added to *prdSet* only after the addition of *Layer 2*, which also adds parameter ShipmentStatus – the only output parameter of W_8 – to *prdSet*. Since, W_8 is not able to produce any parameters that do not already exist in *prdSet* after *Layer 2*, it is rejected by the forward expansion process (Algorithm 2, line 5; see Section 3.4 for more details). Meanwhile, the services that get selected by the process form predecessor-successor relationships among themselves based on their shared input-output parameters. For instance, W_7 (in *Layer 2*) accepts ProductWeight (produced by W_1) and PaymentNumber (produced by W_4 and W_5) as input. Therefore, within the search graph,

Table 6: Services Available for Composition of Shopping Application

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	Order/ Payment	{ProductNumber, ProductPrice, CreditCardBrand, CreditCardNumber}	{ST1234, 75.00, Visa, CCVS56789}	{PaymentNumber}	{PAY1234}	$C_4 = \{CreditCardBrand = Visa\}$
W_5	Order/ Payment	{ProductNumber, ProductPrice, CreditCardBrand, CreditCardNumber}	{ST1234, 75.00, Master, CCMS56789}	{PaymentNumber}	{PAY1234}	$C_5 = \{CreditCardBrand = Master\}$
W_6	Shipment	{PaymentNumber, ProductWeight, ShippingAddress}	{PAY1234, 45, Montreal}	{ShipmentStatus}	{Confirmed}	$C_{61} = \{ProductWeight <= 50\}$ $C_{62} = \{ShippingAddress = Montreal\}$
W_7	Shipment	{PaymentNumber, ProductWeight, ShippingAddress}	{PAY1234, 45, Quebec}	{ShipmentStatus}	{Confirmed}	$C_{71} = \{ProductWeight <= 50\}$ $C_{72} = \{ShippingAddress = Quebec\}$
W_8	Shipment	{PaymentStatus, ProductWeight, ShippingAddress}	{Complete, 45, Canada}	{ShipmentStatus}	{Confirmed}	$C_{81} = \{ProductWeight <= 50\}$ $C_{82} = \{ShippingAddress = Canada\}$

$predecessors(W_7) = \{W_1, W_4, W_5\}$. Figure 10 depicts all such relationships existing within the Shopping application's search graph.

The backward search stage of the composition process uses this search graph to generate service sets that can later be transformed into solution plans. Figure 11 shows the generation of such plan sets when the backtracking begins from *Layer 2*. The parent node of each of the tree-like structures shown in the diagram is an element of the power set of the *Layer 2* service set, i.e., $powerset(\{W_3, W_6, W_7\})$. The middle tier of each tree is composed of the elements of the power set of its parent node's predecessor set belonging to *Layer 1*. For example, Figure 10 shows that $predecessors(W_6) = \{W_1, W_4, W_5\}$, out of which W_4 and W_5 belong to *Layer 1*. Therefore, the second tree in Figure 11 (with $\{W_6\}$ as its parent) has its middle tier composed of $powerset(\{W_4, W_5\})$. Similarly, the lowest tier of each tree is composed of the predecessors of the middle tier services from *Layer 0* (i.e., elements of $powerset(\{W_1\})$).

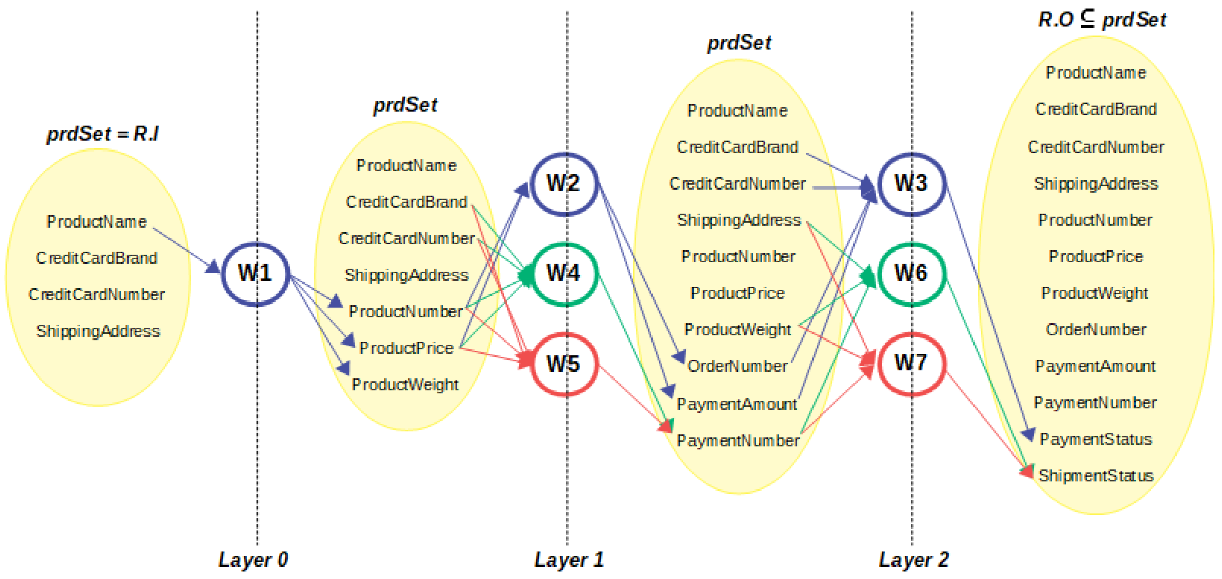


Figure 9: Shopping Service Search Graph Construction

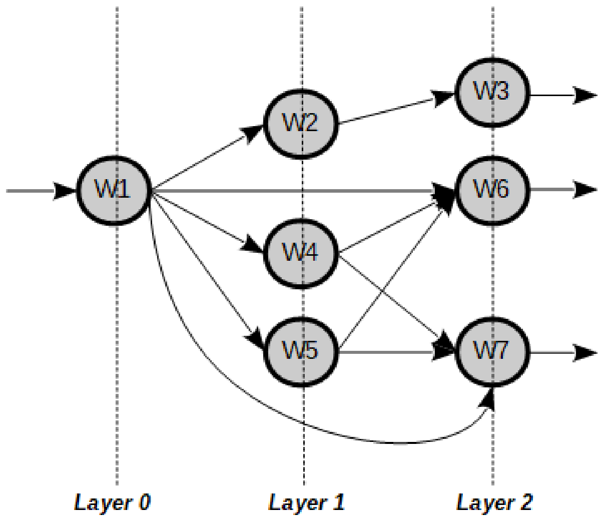


Figure 10: Predecessor-Successor Relationships in Shopping Service Search Graph

The directed edges of each structure follow the backtracking process for each service set from *Layer 2* to *Layer 0*. The sets that are actually constructed are illustrated in green whereas the sets that get discarded during backward search are depicted in red. Set *(1)* is a special case because it would get generated if backward search would be carried out exactly according to Algorithm 4. However, the set would have to be discarded later as it

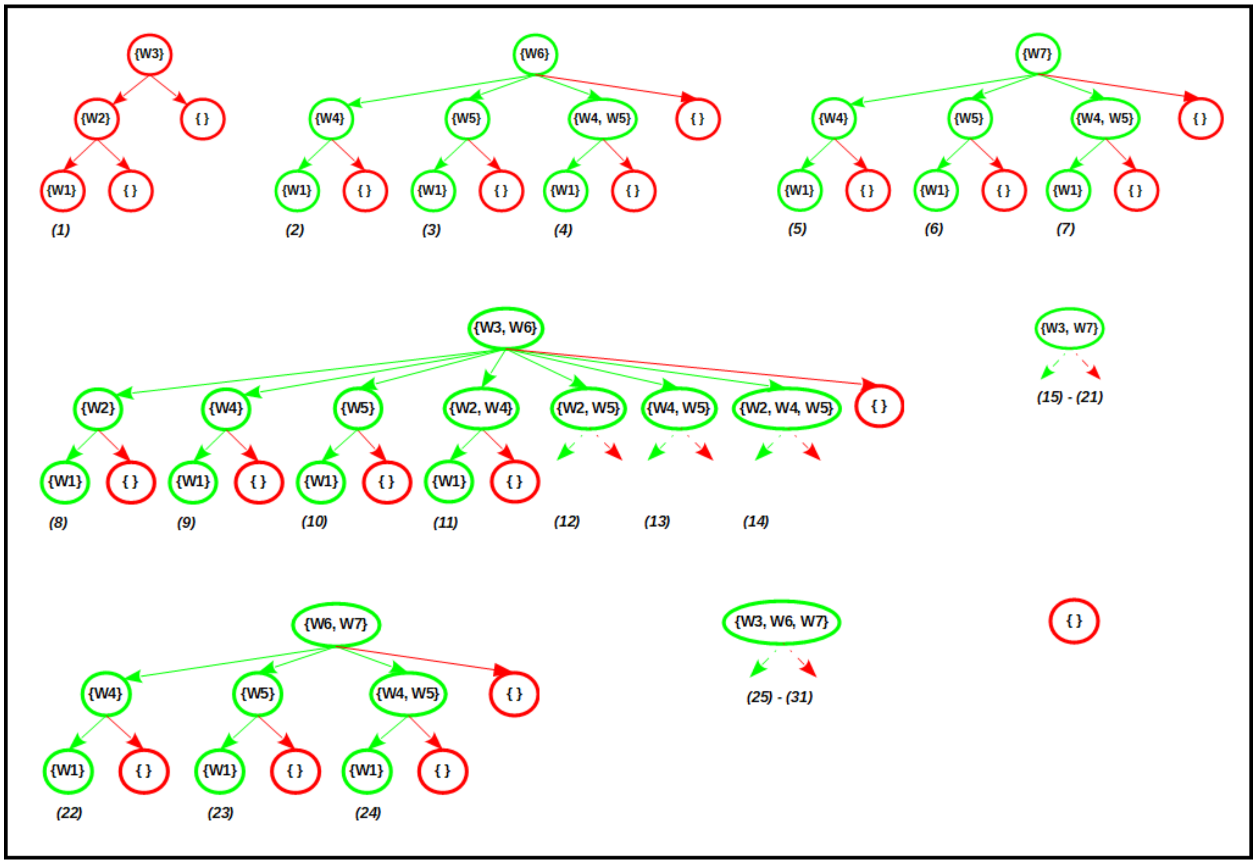


Figure 11: Shopping Service Solution Plan Set Construction

does not produce the goal parameter (ShipmentStatus). In order to save the effort spent on generating and later discarding such sets, one of our optimizations introduced during re-implementation of backward search prevents such sets from being generated in the first place (explained further in Section 3.5.3.2). That is why, set *(1)* is depicted entirely in red in the diagram. In order to avoid redundancy and maintaining clarity in the diagram, sets *(12) - (21)* and *(25) - (31)* have not been extended beyond *Layer 1*. Once all the valid sets illustrated in the figure are generated, the procedure is repeated with *Layer 1* and *Layer 0* as starting layers. However, no valid sets can be extracted from them because, like set *(1)*, they do not produce the goal parameter. Therefore, the procedure has not been depicted here.

The plan sets generated by backward search are validated and pruned by the plan construction stage for generating solution plans for the given composition request. Figure 12

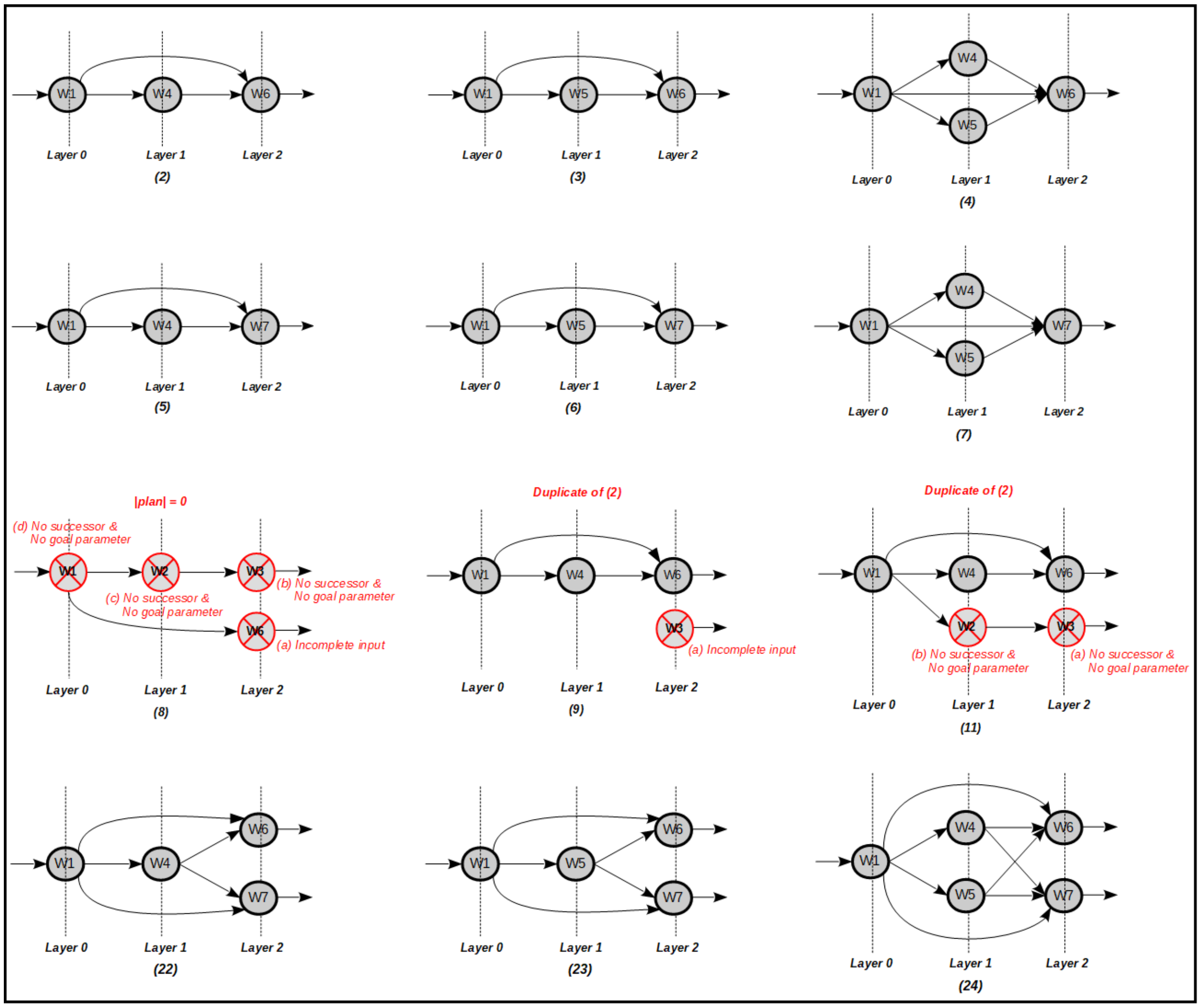


Figure 12: Shopping Service Solution Plan Construction

shows the plan construction for some of the plan sets depicted in Figure 11 (plans are numbered same as their corresponding sets). As an example, consider plan set *(2)*, containing services $\{W_6, W_4, W_1\}$, where, $predecessors(W_6) = \{W_1, W_4, W_5\}$, $predecessors(W_4) = \{W_1\}$ and $predecessors(W_1) = \emptyset$. The services in the set, when arranged in a workflow according to their predecessor-successor relationships, produce solution plan *(2)*, which passes all the validation checks performed by the plan construction stage. Similarly, sets *(3)* - *(7)* and *(22)* - *(24)* also result in generation of valid solution plans for the shopping composition request. However, while plans *(2)*, *(3)*, *(5)* and *(6)* use the minimum number of services necessary for the composition, the other valid plans still contain one or more extraneous services. This limitation of the plan construction process is discussed in detail in Section 6.2.2. Plan sets other than *(2)* - *(7)* and *(22)* - *(24)* either get discarded as invalid or pruning transforms them into duplicates of other valid plans. For example, Figure 12 depicts the sequence in which each service in plan *(8)* fails some validation check during construction, eventually resulting in the plan being rejected for containing no services at all. Meanwhile, plans *(9)* and *(11)* get some of their services removed because of validation failures (shown in figure), which transforms them into plan *(2)* and causes them to be rejected as duplicates by the service composition algorithm (Algorithm 1, lines 8 - 10).

All the valid solution plans generated for the shopping composition request are transformed into constraint-aware plans in which the constraint segment of each service-node comprises of the internal constraints placed on the node's service. For instance, consider the constraint-aware version of solution plan *(5)* shown in Figure 13. Each diamond-circle pair enclosed in a rectangle in the diagram represents a service-node with the diamond standing for constraints and the circle for service. In this plan, the diamonds for W_4 and W_7 hold their respective internal constraints whereas the one for W_1 is blank because the service does not have any internal constraints of its own. Once a constraint-aware plan is constructed, the constraint-adjustment mechanism moves all its constraints to optimal verification points. For instance, considering plan *(5)*, constraints C_4 and C_{72} are imposed on parameters CreditCardBrand and ShippingAddress respectively, which are available as initial parameters from the customer and, therefore, their constraints can be verified before any services in the plan are executed. Consequently, in case any of these constraints fail the run-time verification, no component services would have to be rolled back and no execution effort would be wasted. Therefore, during constraint-adjustment, C_4 and C_{72} are moved to the constraint segment of W_1's service-node (as shown in Figure 14). Similarly, C_{71} constrains ProductWeight parameter whose value is last affected by W_1 and can be verified immediately after W_1 completes execution. Therefore, C_{71} has been moved to the constraint segments of W_1's successor service-nodes. In case it fails, neither W_4 nor W_7 would be executed and only

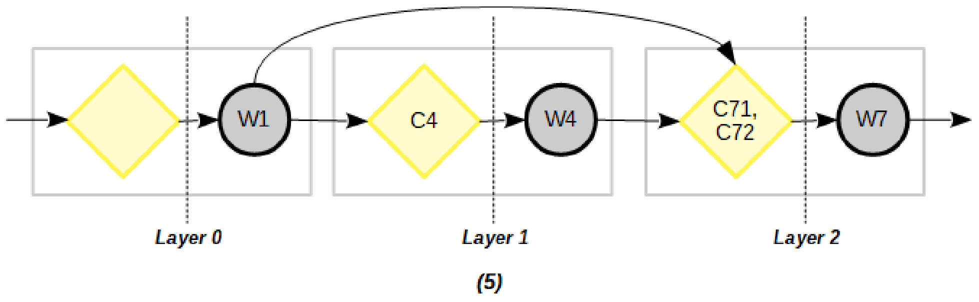


Figure 13: Shopping Service Constraint-Aware Plan before Constraint Adjustment

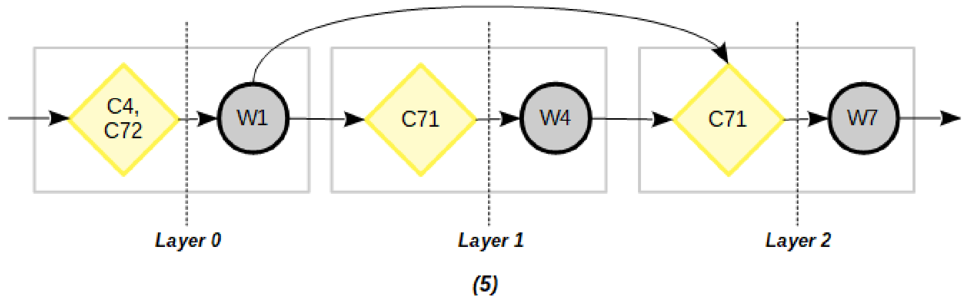


Figure 14: Shopping Service Constraint-Aware Plan after Constraint Adjustment

W_1 would have to be rolled back. In contrast, verification failure of C_{71} in plan *(5)* before adjustment would have forced roll-back of both W_1 and W_4.

In this way, given the shopping composition request and a repository of services as described in Table 6, the service composition process automatically generates nine constraint-aware composition plans with optimally placed internal service constraints.

3.4 Restriction on Service Composition

The planning-graph-based approach to service composition adopted by Laleh et al. [4, 5, 6, 7, 8] places a restriction on the addition of repository services to a search graph, which has a significant impact on the resultant graph and, consequently, on the final constraint-aware composition plans. In this section, we explain this restriction, the rationale behind it and its side-effect.

According to the restriction, a service can be added to a search graph during forward expansion (Algorithm 2) only if it produces at least one parameter as output that does not already exist in the *prdSet* at the time that the service has to be added (lines 5 - 6). The reasons that support this restriction have been listed below:

- **Preventing multiple service occurrences to save execution effort:** Considering a hypothetical search graph, without this restriction, it is possible that the same service, say, W_1, gets added first in, say, *Layer 5*, and then during a later iteration in, say, *Layer 8* of the graph. This can happen if some services in *Layer 7* produce some parameters that can act as inputs to W_1. In case this scenario eventually results in a solution plan that contains both instances of W_1, resources will be unnecessarily spent on executing W_1 twice for computing the same set of parameters.

  To avoid redundant execution of the service, an alternative could be to move W_1 from *Layer 5* to *Layer 8*. However, such a change would affect all the predecessor and successor services of W_1 (even requiring the successors to be moved to *Layer 9* or later). Changes to these services would, in turn, trigger a cascade of service adjustments throughout the graph and ultimately result in a huge and complicated change in its structure, which makes this alternative highly infeasible.

- **Preventing redundant layer construction:** Without this restriction, every time the repository is searched to find services for a new layer, the services that were included in the search graph in an earlier iteration will also be validated again and found eligible to be assigned to the same layers to which they already belong. This is purely redundant processing of services for layers that have already been built, which accounts for a significant amount of processing effort that results in no net growth of the graph.

- **Preventing infinite continuation of forward expansion:** In the absence of this restriction, every eligible service in the service repository, irrespective of whether it is already present in the search graph or not, will be validated and added to the search graph in every iteration. In other words, every iteration will add one or more services to the graph, as a consequence of which the termination condition of forward expansion process – "*until no service could be added to the search graph*" (Algorithm 2, line 14) – will never be satisfied, causing the process to enter an infinite loop once all possible component services for a composition request have been added.

Despite its usefulness, this restriction has a side-effect. It can prevent some of the longer (i.e., containing more layers) yet completely valid solutions to a composition problem from being generated, thereby reducing the final alternative solution count. For instance, consider the search graph depicted in Figure 9. Services in *Layer 2* of the graph generate the requested output parameter ShipmentStatus, which then gets included in *prdSet*. Because of this restriction, Shipment service W_8, despite producing ShipmentStatus as output and having all its predecessors already present in the graph, cannot be added to *Layer 3*, which prevents construction of the alternative solution plan composed of W_1, W_2, W_3 and W_8. Nevertheless, the aim of these composition algorithms or this thesis is not to obtain all possible solutions or the most optimum solution to a composition problem but to be able to generate several alternative constraint-aware solutions within a reasonable amount of time, which can then be translated into Lucid programs for simulation. Therefore, we accept the effects of this restriction as a trade-off between the time complexity of and the solution diversity offered by the planning-graph-based composition methodology.

3.5 Service Composition Implementation

This section discusses the implementation of the service composition algorithms presented in Section 3.2. As already mentioned, these composition algorithms (other than Algorithm 5) as well as the composite service model (discussed in Section 3.1) have been designed by Laleh et al. The primary focus of their research [4, 5, 6, 7, 8] is on effectively introducing the fundamental concepts involved in their composition approach, such as, constraints, context and the co-relation between them. Therefore, their implementation that supports their conceptual model is also specifically focused on and dedicated to simulation of results required for analyzing and evaluating their novel methodologies. Our research, on the other hand, is more concerned with the operational aspects of the service composition process, which serves as a prerequisite for the service translation and simulation process – the primary goal of this thesis. Consequently, we need the composition application to operate as a generic (and not scenario-specific) tool that can accept any valid composition request and service repository and generate a set of possible constraint-aware solution plans in a format that is acceptable as input to the composite service translation framework (discussed in Chapter 4). Additionally, the application also needs to interact with a service repository framework [51] (discussed further with implementation details) that not only provides the readers and writers required to communicate with service repositories written in various formats but also implements some of the fundamental entities of the composition model, including, *Service* and *Constraint*. In order to fulfill these requirements, which cannot be served by Laleh's implementation, we re-implement the composite service model and composition process in JAVA based on Laleh's conceptual design.

Apart from making the application more generic and enabling it to interact easily with other processing units, our re-implementation also serves to complete the sequence of steps involved in the composition process by implementing the *ConstructPlan* algorithm not defined in Laleh's publications (see Section 3.2, Algorithm 5). Furthermore, there are several significant optimizations that we introduce at almost every stage of the composition process to improve the quality of its results, reduce the processing effort involved and enhance its reliability and efficiency, all of which are incorporated into our re-implementation. We explore these optimizations elaborately in Section 3.5.3. Finally, to improve the quality of the composition application from a software engineering point of view, measures, including validation checks, error logging and handling, extensible framework for accepting user input in multiple formats and storage and reuse of composed services, have been taken that make the application more robust, reliable, flexible and maintainable. Sections 3.5.2 and 3.5.4 cover the details of these additional features.

3.5.1 Assumptions

Before we can discuss the characteristics that distinguish our implementation of the service composition approach from the original approach, it is important to clearly state the assumptions upon which we base this implementation. They have been listed below:

- Any intermediate (search graph, plan set etc.) or final (constraint-aware plan) solution generated for a composition problem that comprises of only one service from the service repository is considered invalid since such a solution does not qualify as a composition of (multiple) services.

- Constraints requested by the user, if any, are assumed to be completely satisfied by the composite service(s) generated in response to the request. In order to fully represent the composite service model in our implementation, we allow users to specify constraints as part of their request that the resultant compositions must satisfy. However, user constraints are not the focus of this thesis and, therefore, our current implementation does not verify them. Instead, they are assumed to be satisfied by default for now. Consequently, the $CheckRequesterConstraints$ statement (Algorithm 2, line 7) meant as a placeholder for invoking an algorithm to verify requester constraints is not included in our current implementation (discussed further in Section 6.2.4).

- Services with different names are considered to be different irrespective of their other characteristics, i.e., service name is the unique identifier for services in our implemented system. Two services with the same name cannot be stored in the service repository that we use [51] even if all their other properties – inputs, outputs, constraints and effects – are completely different. Similarly, two services with the same name cannot be included in one search graph, solution plan or constraint-aware plan. This assumption applies to the service repository framework [51], the composition process as well as the composite service translation process.

- During constraint-aware plan construction, all service constraints are assumed to be unique. Even if the constraints represented by two different *Constraint* (JAVA class defined in [51]) objects are exactly the same, they are considered to be different and will be verified separately during composite service simulation/execution. However, multiple copies of the same *Constraint* object are not allowed to be attached to a single service-node in order to reduce redundancy. Therefore, duplicate *Constraint* objects are eliminated during solution plan construction (explored further in Sections 3.5.3.4 and 6.2.3).

3.5.2 Validation Checks

As part of our implementation, we introduce certain validation checks at specific points in the composition process in order to detect erroneous situations as early as possible and save the effort spent on processing entities that would inevitably be discarded at a later stage.

Since a composition request received from a user marks the starting point of a composition process, we perform several checks on it to ensure that it provides all the required information in the expected format before allowing any services to be composed to resolve it. If any of these checks fail, the process is aborted immediately. A composition request is represented as a class (*CompositionRequest*) in our implementation with its objects composed of the same elements as defined in Definition 3. Significant design features of this class followed by the validation checks placed on them have been listed below:

- Each of the requested inputs and outputs is a JAVA *String* consisting of two parts – data type and name – separated by a colon. For instance, "*string : ProductName*". The data types currently handled by the implementation are *int*, *float*, *char*, *boolean* and *string*.

- Although QoS features are not processed by the current implementation, users are allowed to specify them as part of a composition request to ensure completeness of the composition model. Users may provide a list of the names of those QoS features on which they mean to place certain constraints. Acceptable feature names include *COST*, *RESPONSE_TIME*, *RELIABILITY* and *AVAILABILITY* [5].

- All parameter names – input, output, QoS – are case-sensitive. For instance, while *"COST"* is an acceptable QoS parameter, "*Cost*" and "*cost*" are not.

- Same as the QoS features, although user constraints are not processed by our current implementation, they may be specified as part of a composition request for completeness. A user constraint is composed of exactly three elements in the sequence: feature, operator, literal value [5], each separated from the next one by a pipe symbol. For instance, "$int : ProductWeight \mid <= \mid 50$". Each constraint is represented as an object of the *Constraint* JAVA class defined in the service repository framework [51].

- The feature in a user constraint is either a requested input, output or QoS parameter [5], and it follows the same format and naming convention as specified for the parameter used. For instance, while "$string : CreditCardBrand \mid = \mid Visa$" is a valid user constraint, "$CreditCardBrand \mid = \mid Visa$" is not because of the missing data type.

- Acceptable operators for a user constraint include <, >, =, <= and >= (listed in the *Operator* enumeration in [51]).

The validation checks performed on a composition request serve to ensure the following:

- The composition request includes at least one input and at least one output.

- User-requested QoS features follow the expected format as mentioned above.

- User-requested constraints comply with the specifications listed for them above.

Listed below are some additional validation checks that are performed after a valid composition request is successfully created:

- A service repository specified by a user must not be empty, otherwise, the composition process is aborted immediately.

- A composition problem must be solvable based on the given request, repository and algorithms. In other words, at least one solution must be obtained for it at each step of the composition process, i.e., a search graph at the end of forward expansion, at least one plan set at the end of backward search, at least one solution plan at the end of plan construction and so on, otherwise, the composition process is aborted immediately.

- Any intermediate (search graph, plan set etc.) or final (constraint-aware plan) solution generated for a composition problem that comprises of only one service must be discarded as invalid since such a solution does not qualify as a composition of services.

3.5.3 Optimizations

During our re-implementation of the service composition algorithms, we introduce several modifications that can optimize the process by improving the quality of its results, reducing the processing effort involved and enhancing its reliability and efficiency. In this section, we describe these optimizations introduced at every stage of the process together with the rationale behind them or, in other words, the effect that they have on the process.

3.5.3.1 Forward Expansion

Differences between the *ForwardExpansion* (Algorithm 2) and *AddService* (Algorithm 3) algorithms and their implementation that serve to optimize the implemented process are as follows:

- In the *AddService* algorithm, all the inputs of a new service are placed in a *newIn* set (line 1). While checking if an existing service in the search graph can be a predecessor to this new service, the parameters in *newIn* are matched with the outputs of the existing service (line 5). The parameters that match (if any) are then removed from *newIn* (line 8). However, such a removal is not done in the implementation.

  The reason behind this modification is to ***allow more alternative composition solutions*** to be generated. The algorithm approach discards some of the possible alternative solutions to the problem. For instance, consider the Order/Payment services W_4 and W_5 used in the shopping service example in Section 3.3. Both of these services produce the same output parameter PaymentNumber. During the shopping service's search graph construction, $newService$ W_6 is found to accept PaymentNumber as an input, which should allow both W_4 and W_5 to serve as its predecessor, thereby creating two alternative solution branches in the search graph – one containing W_4 and W_6 and the other containing W_5 and W_6. However, with the algorithm approach, after adding W_4 as a predecessor to W_6, PaymentNumber is removed from *newIn* (line 8) because of which W_5 is never added to its predecessor set and the $W_5 - W_6$ branch is never created. The modified implementation, on the other hand, never removes PaymentNumber from *newIn* and, therefore, allows both W_4 and W_5 to serve as predecessors to W_6 and both solution branches to be constructed.

- A search graph generated by forward expansion re-implementation is considered to be valid only if it contains more than one service. No such check is performed in the $ForwardExpansion$ algorithm.

  This additional check helps in ***early elimination of invalid service compositions***. A search graph comprising of only one service (and, consequently, any solution plan extracted from it) does not qualify as a composition of services; it is merely an individual service, which can be retrieved by performing a search on the service repository. Therefore, our implementation discards it as invalid at this early stage and prevents any further effort from being spent on it unnecessarily.

3.5.3.2 Backward Search

Differences between the $BackwardSearch$ algorithm (Algorithm 4) and its implementation that serve to optimize the implemented process are as follows:

- In the $BackwardSearch$ algorithm, every element of every power set of services is processed without any restrictions (lines 3 - 4). However, in the implementation, for a starting layer (i.e., the layer from which backward tracking starts), an element of the power set of the layer's services is processed only if the services in the element produce at least one output parameter requested in the composition request. For instance, consider set *(1)* depicted in Figure 11, which starts with power set element $\{W_3\}$ from *Layer* 2. However, since $\{W_3\}$ does not produce the goal parameter (ShipmentStatus), the element is discarded immediately and plan set *(1)* is never constructed. In contrast, the algorithm approach constructs plan set *(1)* as illustrated in the figure only to reject it later for not producing the goal parameter (lines 16 - 21), thereby wasting the effort spent on the set construction. Same is the case for each element of the power sets of services in *Layer* 1 and *Layer* 0. While the implemented version of backward search refrains from constructing any plan sets for them, the algorithm version constructs each of them only to discard them all later.

Clearly, this additional restriction introduced in our implementation helps ***save the effort spent on processing such invalid branches***. Since, for every set of services in the starting layer, an exponentially-growing branching and backtracking process is triggered, eliminating unnecessary service sets at the very beginning saves a considerable amount of processing effort during backward search. Additionally, preventing such sets from proceeding to the later stages in the process saves the effort involved in pruning the plans constructed from them and discarding duplicate plans that are most likely to result from the pruning, thereby improving the efficiency of the entire service composition process.

It should be noted that this restriction is placed only on the starting layer of a backward search iteration because it represents the last layer of services in a resultant composition plan. If these services do not produce even one of the requested output parameters, they are not serving any requirements and must be discarded as extraneous. However, this logic does not apply to the other layers in the iteration as their service output parameters (even if they are not the goal parameters) might serve as inputs to their successor services in the later layers.

- A plan set generated by backward search re-implementation is considered to be valid only if it contains more than one service. No such check is performed in the *BackwardSearch* algorithm.

This additional check enables ***early elimination of invalid service compositions***. A plan set comprising of only one service (and, consequently, any solution plan extracted from it) does not qualify as a composition of services; it is merely an individual service, which can be retrieved by performing a search on the service repository. Therefore, our implementation discards it as invalid at this early stage and prevents any further effort from being spent on it unnecessarily.

- In the *BackwardSearch* algorithm, a check is performed to ensure that all the inputs of a set of services from the first layer of the given search graph are available as initial parameters in the composition request (lines 9 - 15). However, in our implementation, no such check is performed.

  The purpose of this modification is to ***remove redundant/inapplicable validation checks***. The forward expansion process places only those services in the first layer of a search graph for which all the inputs are available in the composition request. Therefore, another check for the same condition is not required in the backward search process. For the services in other layers, this check is not applicable because they receive their inputs partially or completely from their predecessor services in the preceding layers.

3.5.3.3 Plan Construction

Since the *ConstructPlan* algorithm (Algorithm 5) has been designed as part of this thesis, unlike the other composition algorithms, it does not differ much from our implementation. However, the entire purpose of this algorithm is to optimize solution plans as they are constructed by removing unnecessary services from them and rejecting the plans that are found to be invalid or duplicates of other plans already constructed (as discussed in Sections 3.2 and 3.3 and depicted in Figure 12). Therefore, in this section, we list the optimization operations performed during this stage along with the rationale behind each of them:

- During the construction of a solution plan, we remove two kinds of undesirable services from it: (1) those whose inputs are not completely satisfied by the collection of initial parameters and output parameters of their predecessors present in the plan and (2) those that neither have any successors in the plan nor produce any goal parameters. No such pruning activities are performed explicitly in Laleh's composition approach.

These pruning operations help ***save the processing effort and execution time spent on extraneous component services***. Since these services do not fulfill any requirements, removing them from solution plans at this stage prevents them from being unnecessarily processed during the later stages or executed as part of the resultant composite services.

Pruning also helps ***detect other faults in solution plans***. It shreds the plans down to their minimalistic form, which reveals problems such as being a duplicate of an existing plan, containing less than two component services or not being able to generate all the goal parameters when the plans are subjected to further validation checks during the plan construction phase.

Removal of undesirable services also ***assists with optimized constraint verification***. Along with the unnecessary services, constraints attached to them are also removed from solution plans, thereby saving the effort of adjusting and verifying them in future. Moreover, the chances of constraint verification failure at run-time due to conflicting constraints is also reduced. For instance, consider plan set *(10)* illustrated in Figure 11. The solution plan generated from this set would contain both W_3 and W_5 services, which have $CreditCardBrand = Visa$ and $CreditCardBrand = Master$ as constraints respectively. In the absence of pruning activities, W_3 would not be removed from solution plan *(10)*. Consequently, its resultant constraint-adjusted composite service would require both constraints, C_3 and C_5, to be verified successfully before any of its component services could be executed, which would never be possible and always cause the composition to fail at run-time despite being totally valid.

- A solution plan created during plan construction is considered to be valid only if it contains more than one service. No such check is performed in Laleh's composition approach.

This check enables ***early elimination of invalid service compositions***. A solution plan comprising of only one service does not qualify as a composition of services; it is merely an individual service, which can be retrieved by performing a search on the service repository. Therefore, our implementation (as well as algorithm) discards it as invalid at this stage itself and prevents any further effort from being spent on it unnecessarily.

- For a solution plan to be valid, all its component services must collectively be able to generate all the goal (i.e., requested output) parameters as output. No such restriction is explicitly imposed on solution plans in Laleh's composition approach.

  The purpose of this restriction is to ***ensure complete fulfillment of composition requests*** by the solutions generated for them. Solution plans that do not generate all of the requested output parameters result in composite services that are unable to fulfill user requirements, which defeats the entire purpose of the composition process. Hence, such plans are discarded at this stage so that no effort is spent on further processing them unnecessarily.

3.5.3.4 Constraint-aware Plan Construction

Differences between our implementation and the algorithms (and, by extension, the original implementation) for constraint-aware plan construction (Algorithm 1, lines 13 - 21) and constraint adjustment (Algorithm 6) that serve to optimize the re-implemented process are as follows:

- During the plan construction stage (Algorithm 5), plans are subjected to certain pruning activities. Because of this, in the resultant plans, some service-node objects end up containing predecessor and successor lists with pointers to service-nodes that no longer exist in the said plans. For instance, consider solution plan *(11)* depicted in Figure 12, in which W_2 – a successor of W_1 – gets removed from the plan during construction, leaving an unnecessary pointer to W_2 in the successor list of W_1. In our implementation, during transformation of a solution plan into a constraint-aware plan, such pointers, if found, are removed from the predecessor and successor lists of its constituent service-nodes.

This measure ***prevents errors from occurring due to missing service-nodes***. If pointers to the service-nodes that have been eliminated during pruning are not removed from plans during this transformation, the later stages of service composition could result in errors while attempting to process such missing nodes as part of a plan's predecessor and successor lists.

Additionally, it helps ***reduce the effort spent on processing deleted service-nodes***. Retaining pointers to service-nodes that have been eliminated during pruning could still trigger node-level iterations, if not throw errors, at every upcoming stage in service composition. Since these service-nodes would not be relevant to the plan in question, the processing involved in even traversing through these nodes (without further action) would be unnecessary and could accumulate to a substantial wastage of processing effort for larger plans.

- Due to the pruning performed in the plan construction stage (Algorithm 5), some resultant solution plans might contain empty service layers. In our implementation, during transformation of a solution plan into a constraint-aware plan, empty layers are removed from the plan and layer indexes of all the service-nodes in the resultant plan are adjusted according to the updated layer sequence.

  This pruning activity facilitates ***removal of unnecessary information from and clearer representation of constraint-aware plans***. Since an empty layer does not contain any service-nodes, it carries no relevant information about a plan that would need to be processed. Therefore, in order to optimize the plan, this extraneous information is removed from it. Additionally, removal of unnecessary empty layers enables a constraint-aware plan (i.e., composite service) to be represented more clearly and efficiently when it is translated into other formats during the translation/simulation phase (discussed in Chapter 4).

This optimization also ***saves the effort spent on processing empty service layers***. Although an empty service layer would not trigger any intensive processing, its presence still invokes a layer-level iteration at every stage of service composition. For large composition plans that could easily contain several empty layers, these iterations could consume a considerable processing effort in an already expensive process.

- As part of our constraint-adjustment implementation, a *Constraint* object is assigned to a service-node only once. This decision is based on the assumption that distinction between constraints is made based on their JAVA objects and not on their constituting elements – feature, operator and literal value (discussed further in Sections 3.5.1 and 6.2.3).

  This approach helps ***reduce the effort spent on processing duplicate constraints*** to some extent. Designing a solution for completely eliminating duplicate constraints from a service-node is a complex problem in itself and is not the focus of this thesis. However, at the very basic level, in an effort to avoid processing the same constraint multiple times for the same service-node, we prevent duplicate *Constraint* objects from being assigned to any service-node of a constraint-aware plan.

3.5.3.5 Service Composition

Differences between our implementation and the algorithm (and, by extension, the original implementation) for *ServiceComposition* (Algorithm 1) that serve to optimize the re-implemented process are as follows:

- According to the *ServiceComposition* algorithm, a composition request and a set of available services are provided as input to the process. However, in our implementation, a composition request configuration object (explained in Section 3.5.4.2) and a logger object (discussed in Section 3.5.4.3) are provided as arguments to the composition method.

  This modification ***facilitates storage and re-use of composite services***. The request configuration object contains the inputs, outputs, QoS features and constraints requested by the user, which are used to create a *CompositionRequest* object (described in Section 3.5.2) before triggering the various composition stages. It also holds the location of a service repository file which can be parsed to extract the available services. Besides these, the configuration contains a flag, which, when set to "Y" by the user, causes the constraint-aware plans constructed to resolve the given request to be stored as composite services in the given repository. This is an additional feature of our implementation, which allows composite services to be stored and re-used as components for future compositions (detailed in Sections 3.5.4.2 and 3.5.4.6).

Furthermore, creation of a configuration object ***allows our application to use different sources for user input without affecting the composition process***. The object is not only essential for supplying any additional information, such as, the storage flag, required by the composition process but also acts as a generic source of information for the core process that hides the actual medium through which the information is gathered from the user, be it the console or a file or any other sources that might be added to the existing architecture in future (explained in Section 3.5.4.4), thereby promoting lower coupling between the user-input and service composition units.

Meanwhile, the logger object ***enables logging of error/status messages*** generated throughout our application in a text file for reference (discussed in Section 3.5.4.3).

- The *ServiceComposition* algorithm does not include any specific validation checks. However, in our implementation, several checks are performed on the composition request, service repository and results of the various stages of the service composition process (described in Section 3.5.2).

  These validation checks help ***minimize the processing effort wasted in case of failures***. They ensure that the composition process continues after completing each step only if all the prerequisites for the next step are satisfied and if it is worth triggering the next step so as to minimize the effort already spent in case a failure occurs.

- The *ServiceComposition* algorithm includes transformation of solution plans into constraint-aware plans (lines 13 - 21). However, in our implementation, constraint-aware plan construction and constraint-adjustment are written as a single separate unit that is invoked by the service composition unit whenever required just like the other composition stages. Similarly, the check for adding only unique plans to the list of valid solution plans (lines 8 - 10) is incorporated in the plan construction stage (not the service composition unit) of our implementation. Additionally, our implementation of the *ServiceComposition* algorithm does not include the loops (lines 3 - 11 and 13 - 21) placed around statements that invoke the *BackwardSearch*, *AdjustConstraint* and *ConstructPlan* algorithms. These loops are instead included in the implementation of the invoked algorithms themselves.

  The purpose of these structural modifications is to ***improve modularity and maintainability of and reduce inter-unit coupling within our application***. Since the complete implementation of each algorithm is now contained within its own class, any future modifications in these sub-processes (if required) would not affect the implementation of the other units.

3.5.4 Additional Features

In order to make our service composition implementation more flexible and to enable its use as a tool/application, we introduce some additional functionality into it that is not found in the original implementation (by Laleh) of the composition algorithms presented in Section 3.2. In this section, we describe those additional features and the related architecture.

3.5.4.1 Service Composition Driver

The service composition driver is responsible for prompting the user to provide the inputs required for executing the service composition process, for triggering the various stages involved in the process in the proper sequence and for displaying the final status (success/failure) of the process on the console. More specifically, the driver performs the following tasks:

1. Prompt the user on the console to select a mode of input for providing details of the composition request configuration. At present, the user can select from console and XML file modes, although the architecture in place allows these options to be extended to other modes as well (discussed in Section 3.5.4.2).

2. If XML file mode is selected, prompt the user to provide XML configuration file path.

3. Depending on the selected mode of input, invoke the corresponding request configuration reader to fetch the composition request and other necessary information from the user and use it to create a request configuration object (described in Section 3.5.4.2).

4. Create a logger object (described in Section 3.5.4.3), which would be passed across methods to allow error messages generated throughout the process to be recorded in a log file.

5. Trigger the service composition process, passing it the request configuration and logger objects created.

6. If the composition process is successful, write the constraint-aware plans generated into a text-based plans file. However, if the composition process fails at any point, display a failure message and invite the user to check the log file for error details.

For example, Listing 3.1 shows the contents of the XML configuration file representing the composition request for constructing the online shopping service discussed in Section 3.3. Its corresponding set of available services, as described in Table 6, can be represented as an XML repository file part of whose contents are displayed in Listing 3.2. The nine constraint-aware plans that get generated based on these request configuration and repository files are written to a plans file titled *plans.txt* (depicted partly in Listing 3.3). In an alternate scenario, if the given service repository file does not contain any component services, a log file same as the one depicted in Listing 3.4 gets generated instead of a plans file.

Listing 3.1: Shopping Service Request Configuration (in XML File Format)

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<requestconfig>
    <inputs value="string : ProductName, string : CreditCardBrand,
                   string : CreditCardNumber, string : ShippingAddress"/>
    <outputs value="string : ShipmentStatus"/>
    <qos value=""/>
    <constraints value=""/>
    <repofilename value="testinput/servicerepos/Services_Repo_Shopping.xml"/>
    <storecsflag value="N"/>
</requestconfig>
```

Listing 3.2: Available Services for Shopping Composition (in XML File Format)

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<services>
    <service name="W1">
        <inputs>
            <instance name="string : ProductName"/>
        </inputs>
        <outputs>
            <instance name="string : ProductNumber"/>
            <instance name="float : ProductPrice"/>
            <instance name="int : ProductWeight"/>
        </outputs>
        <constraints>
        </constraints>
        <effects>
            <instance name="string : ProductNumber"/>
            <instance name="float : ProductPrice"/>
            <instance name="int : ProductWeight"/>
        </effects>
    </service>
    <service name="W2">
     ...
    </service>
    <service name="W3">
        <inputs>
            <instance name="string : OrderNumber"/>
            <instance name="float : PaymentAmount"/>
            <instance name="string : CreditCardBrand"/>
            <instance name="string : CreditCardNumber"/>
        </inputs>
        <outputs>
            <instance name="string : PaymentStatus"/>
        </outputs>
        <constraints>
            <instance>
                <servicename name="W3"/>
                < literalvalue  name="Visa"/>
                <type name="string : CreditCardBrand"/>
                <operator name="="/>
            </instance>
        </constraints>
        <effects>
            <instance name="string : PaymentStatus"/>
        </effects>
    </service>
     ...
</services>
```

Listing 3.3: Shopping Service Constraint-aware Plans (in Text File Format)

```
Plan 1:
 ...

Plan 2:
Layer 0: {} [string : CreditCardBrand EQUALS Master, string : ShippingAddress
EQUALS Quebec] W1 {W5, W7}
Layer 1: {W1} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W5 {W7}
Layer 2: {W1, W5} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W7 {}

Plan 3:
Layer 0: {} [string : CreditCardBrand EQUALS Visa, string : ShippingAddress
EQUALS Quebec] W1 {W4, W7}
Layer 1: {W1} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W4 {W7}
Layer 2: {W1, W4} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W7 {}

Plan 4:
 ...

Plan 5:
Layer 0: {} [string : CreditCardBrand EQUALS Visa, string : ShippingAddress
EQUALS Montreal] W1 {W4, W6}
Layer 1: {W1} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W4 {W6}
Layer 2: {W1, W4} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W6 {}

Plan 6:
Layer 0: {} [string : CreditCardBrand EQUALS Master, string : ShippingAddress
EQUALS Montreal] W1 {W5, W6}
Layer 1: {W1} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W5 {W6}
Layer 2: {W1, W5} [int : ProductWeight LESS_THAN_OR_EQUAL_TO 50] W6 {}

```

```
Plan 7:
 ...

Plan 8:
 ...

Plan 9:
 ...
```

Listing 3.4: Shopping Composition Empty Repository Log (in Text File Format)

```
Service repository is empty.
Aborting service composition process.
```

3.5.4.2 Composition Request Configuration and Readers

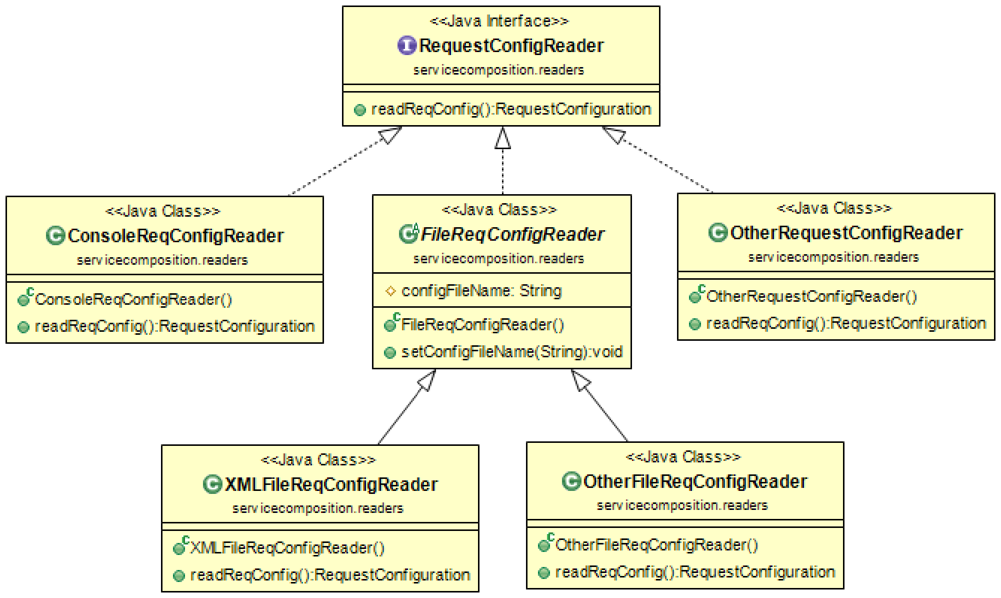
A *RequestConfiguration* object contains all the information provided by the user that is required to execute the service composition process. It consists of the following elements:

- **Inputs:** Comma-separated list of inputs that the customer can provide
- **Outputs:** Comma-separated list of outputs expected by the customer
- **QoS:** Comma-separated list of QoS features expected from the service by the customer
- **Constraints:** Comma-separated list of constraints imposed by the user/requester
- **Repository Filename:** Complete path of the file that contains the available services
- **Composite Service (CS) Storage Flag:** Single-character flag, which, when set to "Y", causes the constraint-aware plans constructed as solutions to the given request to be transformed into layered composite service objects (described in Section 3.5.4.6) and stored back in the given service repository file. When set to "N", this flag prevents the composition solutions from being stored in the given repository. At present, this flag works only for serialized JAVA object repositories.

Once created and passed to the service composition triggering procedure, the inputs, outputs, QoS and constraints from the *RequestConfiguration* object are used to create a

CompositionRequest object (described in Section 3.5.2) while the repository file path is used to locate and parse the file and extract a list of services (or *Service* objects) available for composition. These objects are then used for executing the composition process. If any solutions to the composition problem are successfully constructed, value of the CS storage flag is inspected to decide whether or not to store the solutions in the given repository.

As mentioned in Section 3.5.4.1, a user can opt for different modes for supplying the composition request configuration details. We have designed and implemented an architecture (as depicted in Figure 15) that allows modular addition and removal of readers for each of these modes. At present, this implementation can support console and XML file readers. Major design and implementation specifications of this architecture have been listed below:



<<Java Interface>>
RequestConfigReader
servicecomposition.readers
readReqConfig():RequestConfiguration
<<Java Class>>
ConsoleReqConfigReader
servicecomposition.readers
ConsoleReqConfigReader()
readReqConfig():RequestConfiguration
<<Java Class>>
FileReqConfigReader
servicecomposition.readers
configFileName: String
FileReqConfigReader()
setConfigFileName(String):void
<<Java Class>>
OtherRequestConfigReader
servicecomposition.readers
OtherRequestConfigReader()
readReqConfig():RequestConfiguration
<<Java Class>>
XMLFileReqConfigReader
servicecomposition.readers
XMLFileReqConfigReader()
readReqConfig():RequestConfiguration
<<Java Class>>
OtherFileReqConfigReader
servicecomposition.readers
OtherFileReqConfigReader()
readReqConfig():RequestConfiguration

Figure 15: Composition Request Configuration Reader Architecture

- **RequestConfigReader** is the interface to be implemented by all concrete composition request configuration readers. It declares the *readReqConfig* method, which should be defined to accept request configuration details from the user based on the mode of input being handled by each concrete reader.

- **ConsoleReqConfigReader** is the concrete composition request configuration reader for interacting with the user through the console to obtain request configuration details. It implements the *RequestConfigReader* interface and defines the *readReqConfig* method. After reading all the required information, the method creates a *RequestConfiguration* object to be used for service composition.

- **FileReqConfigReader** is the abstract class to be extended by all concrete composition request configuration file readers. It implements the *RequestConfigReader* interface but does not define the *readReqConfig* method; the method is expected to be defined by the concrete file readers.

  This class contains a *configFileName* data member and a mutator method for assigning a value to it. The *configFileName* member is inherited by all concrete file readers and stores the complete path of the file to be read by them.

- **XMLFileReqConfigReader** is the concrete composition request configuration reader for extracting request configuration details from a user-specified XML file. It extends the *FileReqConfigReader* class and defines the *readReqConfig* method. After reading all the required information, the method creates a *RequestConfiguration* object to be used for service composition.

Readers for other file formats (depicted as *OtherFileReqConfigReader* in Figure 15) can be easily added to the existing architecture by extending the *FileReqConfigReader* class and defining the *readReqConfig* method to do the file-specific parsing. To include readers for other modes of input, such as, databases, a new class (depicted as *OtherRequestConfigReader* in Figure 15) can be added to this hierarchy and made to implement the *RequestConfigReader* interface while defining the required behavior in the *readReqConfig* method.

The *readReqConfig* method for the console reader consecutively prompts the user to provide the value for each element of a request configuration according to the following rules:

- Values for the inputs, outputs, QoS and constraints must follow the same formats as those defined for the elements of a composition request in Section 3.5.2.

- A repository filename must include the complete name (with extension) and path of the repository file. For now, only *.txt* and *.xml* are recognized as acceptable file extensions.

- Acceptable values for the CS storage flag are "Y" or "y" to allow storage and "N" or "n" to prevent storage.

When there are no values to be provided for an element, such as, for the optional QoS element, pressing the *Return* key when prompted for the element's value allows the user to skip to the next element directly.

The *readReqConfig* method for the XML file reader, on the other hand, parses an XML configuration file, which must comprise of the following elements:

- **requestconfig:** It is the root element of the XML file.

- **inputs, outputs, qos, constraints:** Each of these elements appear once in the XML file, as a sub-element of the root. The value assigned to their "value" attribute is a comma-separated list of requested inputs, outputs, QoS features and constraints respectively.

- **repofilename:** It appears once in the XML file, as a sub-element of the root. The value assigned to its "value" attribute is the complete name (with extension) and path of the repository file.

- **storecsflag:** It appears once in the XML file, as a sub-element of the root. Acceptable values for its "value" attribute are "Y" or "y" to allow storage and "N" or "n" to prevent storage.

The same format and rules apply to request configuration's XML elements as defined for the console mode. If there are no values to be provided for an element, such as, for the optional constraints element, the double quotes for that element's value may be left empty.

3.5.4.3 Logger

We have included a simple message logging utility in our implementation, which is shared by the service composition and translation (discussed in Chapter 4) processes. It allows all the error and status messages generated during a process to be recorded in a text file. Each time a new process is launched, a new logger object is created by the process driver (see Section 3.5.4.1). Each logger object is associated with a specific text file, which is opened for writing in "append" mode so that passing the same logger object across the methods that get called during a process records all generated messages in the same text file. This helps in generating a persistent error record of a composition/translation run and also assists in performing automated unit testing by eliminating the need for console interaction.

3.5.4.4 Service Repository Parser Alternatives

Depending on the type (file extension) of the service repository file whose details are provided in the composition request configuration (described in Section 3.5.4.2) by the user, a suitable constrained service parser (borrowed from [51]) can be employed by our service composition process to extract component service specifications from the given constrained service repository. Currently, a serialized JAVA object parser is used to parse *.txt* files holding serialized *Service* objects and an XML parser is used to parse *.xml* files listing service definitions in a custom format (defined in [51]). This functionality can be easily extended to include as many repository file types as there are parsers available in the service repository framework [51], thereby making the composition application more versatile.

3.5.4.5 Plans File

For a composition request that can be successfully served based on the information provided by the user, our service composition implementation generates a text file containing descriptions of all constraint-aware plans that can serve the given request. Contents of this plans file have been described below:

- All constraint-aware plans generated for a composition request are numbered and listed in the file in sequence.

- Each plan consists of layers, with their indexes starting from 0 and increasing by 1 for each consecutive layer. A plan description (listed in the file) comprises of a list of descriptions of all its layers, each on a new line, in increasing order of their layer index.

- A layer description describes the service-nodes that constitute that layer. Each node description is separated from the next by a comma. The order of service-nodes (or their description) within a layer is not important.

- A service-node description consists of four parts: predecessor names, constraints, service name and successor names. It is formatted as:
  "{*predecessor names*} [*constraints*] *service name* {*successor names*}"

- *predecessor names* for a service-node are a comma-separated list of service names of the service-nodes that act as predecessors to the given service-node. The order of names in the list is not important.

- *constraints* for a service-node are a comma-separated list of constraints that must be satisfied before the service contained within the service-node can be executed. These may include the internal constraints placed on the enclosed service itself and/or the constraints that are transferred to the service-node from other nodes in the plan during constraint adjustment (Algorithm 6). The order of constraint descriptions in the list is not important. Each constraint description is formatted as:
  "featuredatatype : featurename operatorname literalvalue", where:

  - *featuredatatype* is the data type of a component service input parameter and may be *int*, *float*, *char*, *boolean* or *string*.
  - *featurename* is the name of the input parameter.
  - *operatorname* may be *LESS_THAN*, *GREATER_THAN*, *EQUALS*, *LESS_THAN_OR_EQUAL_TO* or *GREATER_THAN_OR_EQUAL_TO*.

  - *literalvalue* is the value to which the feature's value will be compared during constraint verification.

- *service name* for a service-node is the name of the service encapsulated within the node.
- *successor names* for a service-node are a comma-separated list of service names of the service-nodes that act as successors to the given service-node. The order of names in the list is not important.
- Service-nodes in the first layer do not have any predecessors while those in the last layer do not have any successors. In such cases, the curly braces that enclose the predecessor/successor list are left empty ({}). Similarly, service-nodes that do not have any constraints have their enclosing square brackets empty ([ ]).
- Since every service-node must have a service, and every service must have a name, therefore, a service-node description always has a service name.

3.5.4.6 Layered Composite Service Storage and Reuse

We have added a layered composite service decorator (called *LayeredCompositeService*) to the existing service repository framework [51]. A utility class has also been defined within our service composition implementation that uses the decorator to create a layered composite service object for each of the constraint-aware plans created as a solution to a composition request. This utility also enables appending this composite service object to the service repository from which the component services for its construction are extracted. The stored composite service can then be used as a component service for future compositions. The limitation to this feature is that, at present, the storage and reuse of composite services is restricted to serialized Java object repositories only. However, it can be extended to other repository formats by designing proper representations, parsers and writers for them.

Elements of the composite service object created by this utility are listed below:

- **Composite Service Name:** “CompSvc_” concatenated with the system time (in nanoseconds) at which the object is created.
- **Composite Service Inputs:** List of composition request inputs.
- **Composite Service Outputs:** List of composition request outputs.
- **Composite Service Effects:** Set of all the effects of all the services contained within the service-nodes that constitute the constraint-aware plan.
- **Composite Service Constraints:** Set of all the constraints of all the service-nodes that constitute the constraint-aware plan. This information enables optimum constraint adjustment in the plan (if any) that uses this composite service as a component.
- **Constraint-aware Plan:** The constraint-aware solution plan that the composite service object represents.

In order to create a composite service object, the utility class first invokes the constrained service decorator from the service repository framework to create a simple constrained service with the composite service’s name, inputs, outputs, effects and constraints as data members. Then, it invokes our layered composite service decorator to decorate this constrained service with the constraint-aware plan (instance of *ConstraintAwarePlan* class) to form a layered composite service object.

3.6 Summary

In this chapter, we explained the unique planning-graph-based service composition and constraint-adjustment approach devised by Laleh et al. along with the improvements that we introduce into it. This improved approach is used in our research for generating constraint-aware composite services that can be translated into OBJECTIVE LUCID programs intended for execution on GIPSY. Having examined the process of constructing composite services in elaborate detail in this chapter, in the next chapter, we present a similar discussion on the translation of these services into various useful formats, including OBJECTIVE LUCID programs capable of being simulated/executed on GIPSY, which forms the penultimate stage in our composite service verification process (as outlined in Section 1.5).

Chapter 4

Composite Service Translation

In Chapter 3, we present a detailed discussion on the process of generation of constraint-aware composite web services for a given composition request. In order for these services to be simulated/executed on GIPSY, as per the composite service verification procedure described in Section 1.5, they must first be translated into some dialect of Lucid – the only language that can be interpreted by GIPSY. Based on the reasoning presented in Section 2.1.4, Objective Lucid proves to be the best candidate for this task. Therefore, continuing with the explanation of the service verification procedure, in this chapter, we describe the extensible framework that we have designed for translating constraint-aware composite web services (as defined in Chapter 3) into various different formats, elaborating specifically on the translation to Objective Lucid. Additionally, we also present a comparatively brief discussion on the other translation modules that we have already designed and plugged-in to the framework together with the implementation features that make the application more flexible and maintainable.

4.1 Composite Service Translator Framework

For translating the layered composite services generated by the service composition approach discussed in Chapter 3, we have designed and implemented an extensible translator framework capable of allowing modular plugging-in and -out of different translator programs, as required. Each of these modules can translate a layered composite service into a specific target language/model, thereby making it possible for us to utilize their unique qualities not only for enhancing the visualization, readability and, hence, understandability of complex compositions (such as, through DOT graph and XML) but also for augmenting our verification system with specialized analysis and validation capabilities (such as those offered by Petri nets) in future. At present, our translator framework consists of modules that can support translation of layered composite services into OBJECTIVE LUCID programs, XML files and DOT graphs. Major design and implementation specifications of its architecture (as depicted in Figure 16) have been listed below:

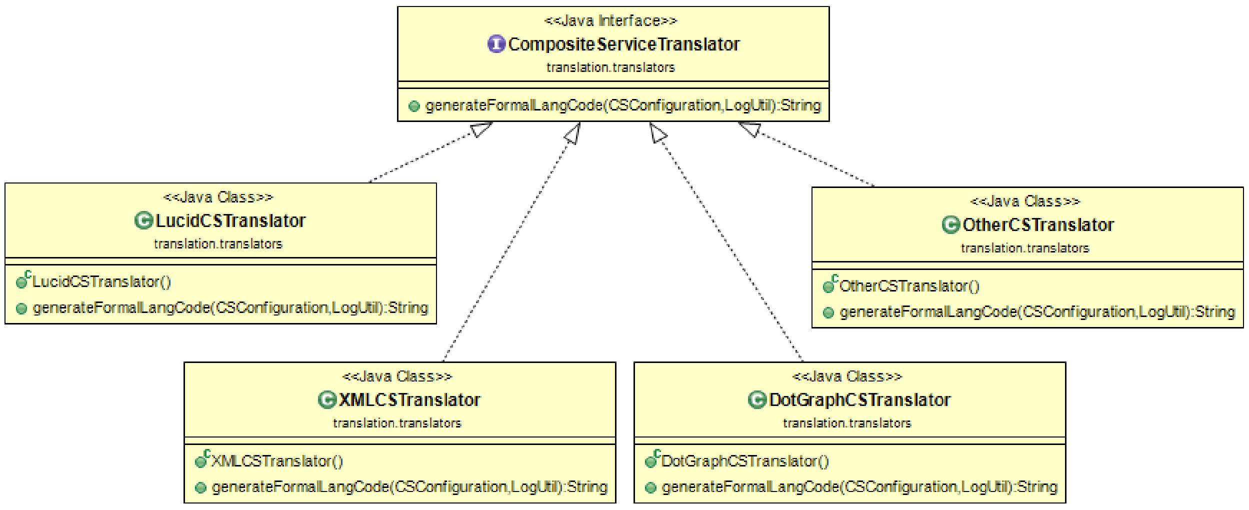


Figure 16: Composite Service Translator Architecture

- **CompositeServiceTranslator** is the interface to be implemented by all concrete composite service translators. It declares the *generateFormalLangCode* method, which should be defined to translate a given *LayeredCompositeService* object (described in Section 3.5.4.6) into a specific formal language based on the target language of each concrete translator.

  A *CSConfiguration* object (described in Section 4.3.2) containing all the information required for performing a composite service translation and a logger object (described in Section 3.5.4.3) for recording any error messages generated during the translation are provided as input parameters to the *generateFormalLangCode* method.

- **LucidCSTranslator** is the concrete translator for generating the OBJECTIVE LUCID program equivalent of a given *LayeredCompositeService* object (following the procedure detailed in Section 4.2). It implements the *CompositeServiceTranslator* interface and defines the *generateFormalLangCode* method, making it responsible for performing the following tasks:
  - Since the OBJECTIVE LUCID programs generated by this translator are ultimately meant to be executed on GIPSY for composite service verification, they need to be provided with values for the inputs required by the composite service. These values are fetched from the user during creation of the *CSConfiguration* object that is supplied as input to the *generateFormalLangCode* method. Before triggering the translation, the method performs some basic validation checks on the given input values to ensure compliance of data types and other specifications (described in Section 4.2). Only if all the validation checks are successful, the translation process is allowed to proceed further, otherwise, it is immediately terminated in error, recording details of the failure in a log file using the given logger object.
  - Once all the validation checks are cleared, the *LayeredCompositeService* object to be translated and the composite service's input details are extracted from the given *CSConfiguration* object and used to generate the JAVA and OBJECTIVE LUCID code segments for the given composite service, which are then merged together to form its equivalent OBJECTIVE LUCID program.
  - The generated program is written into a *.ipl* file with the composite service's name preceded by "CSLucid_" as its title and stored in the destination location obtained from the given *CSConfiguration* object. For successful translations, the *generateFormalLangCode* method terminates by returning the complete name

(with extension) and location of the file to which the program is written. In case any failure occurs during the translation process, *null* is returned by the method and relevant error messages are recorded in the log file associated with the given logger object.

As an example, consider the layered composite service depicted in Figure 17 and its equivalent OBJECTIVE LUCID program shown in Listing 4.1 (explained in Section 4.2). It should be noted here that this is the same composite service designed for calculating the range of three numbers that is used as an example in Section 2.1.

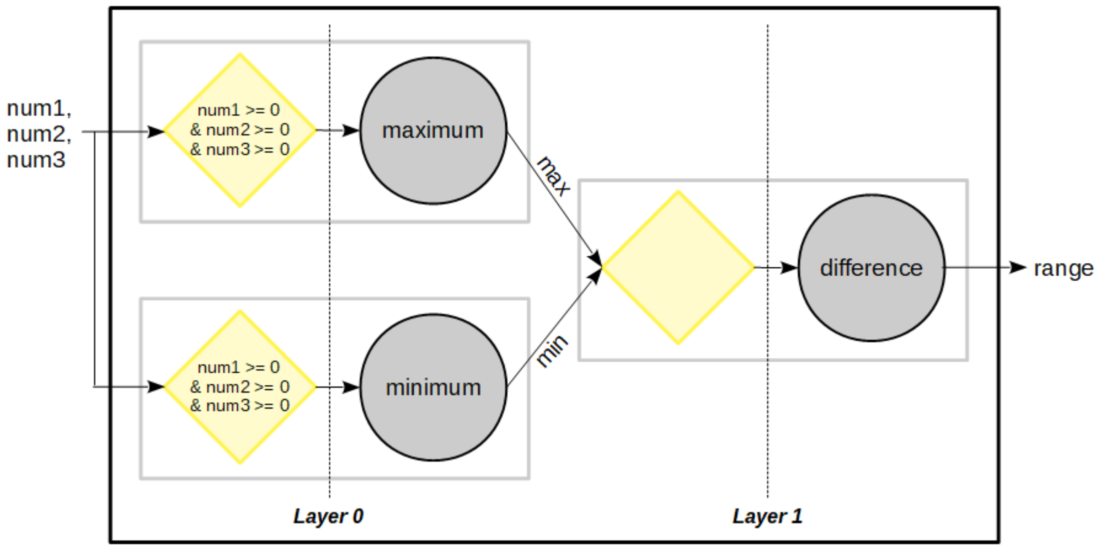


Figure 17: Range Layered Composite Service

Listing 4.1: OBJECTIVE LUCID Translation of Range Composite Service

```
#JAVA
public class CAWSReqComp
{
    private int diff;

    public CAWSReqComp(int diff)
    {
        this.diff = diff;
    }
}

```

```
public class CAWSdifference
{
    private int max;
    private int min;
    private int diff;

    public CAWSdifference(int max, int min)
    {
        this.max = max;
        this.min = min;
        diff = 0;
    }

    public void process()
    {
        diff = 10;
    }
}

public CAWSdifference difference(int max, int min)
{
    CAWSdifference oCAWSdifference = new CAWSdifference(max, min);
    oCAWSdifference.process();
    return oCAWSdifference;
}

public class CAWSmaximum
{
    private int num1;
    private int num2;
    private int num3;
    private int max;

    public CAWSmaximum(int num1, int num2, int num3)
    {
        this.num1 = num1;
        this.num2 = num2;
        this.num3 = num3;
        max = 0;
    }

    public void process()
    {
        max = 10;
    }
}

public CAWSmaximum maximum(int num1, int num2, int num3)
```

```
{
    CAWSmaximum oCAWSmaximum =new CAWSmaximum(num1, num2, num3);
    oCAWSmaximum.process();
    return oCAWSmaximum;
}

public class CAWSminimum
{
    private int num1;
    private int num2;
    private int num3;
    private int min;

    public CAWSminimum(int num1, int num2, int num3)
    {
        this.num1 = num1;
        this.num2 = num2;
        this.num3 = num3;
        min = 0;
    }

    public void process()
    {
        min = 10;
    }
}

public CAWSminimum minimum(int num1, int num2, int num3)
{
    CAWSminimum oCAWSminimum = new CAWSminimum(num1, num2, num3);
    oCAWSminimum.process();
    return oCAWSminimum;
}

#OBJECTIVELUCID
oCAWSMain @.g_num1 10 @.g_num2 12 @.g_num3 14
where
    dimension g_num1, g_num2, g_num3;

    oCAWSMain = CAWSReqComp(#.l_diff)
                wvr CAWSReqCnstr
                @.l_diff oCAWSdifference.diff
                where
                    dimension l_diff ;
                    CAWSReqCnstr = true;

                    oCAWSdifference = difference (#.l_max, #.l_min)
                                      wvr c_difference
```

```
                                               @.l_max oCAWSmaximum.max
                                               @.l_min oCAWSminimum.min
                                               where
                                                   dimension l_max, l_min;
                                                    c_difference  = true;
                                               end;

                         oCAWSmaximum = maximum (#.l_num1, #.l_num2,
                                                     #.l_num3)
                                               wvr c_maximum
                                               @.l_num1 #.g_num1
                                               @.l_num2 #.g_num2
                                               @.l_num3 #.g_num3
                                               where
                                                   dimension l_num1, l_num2, l_num3;
                                                   c_maximum = #.l_num1 >= 0 and
                                                               #.l_num2 >= 0 and
                                                               #.l_num3 >= 0;
                                               end;

                         oCAWSminimum = minimum (#.l_num1, #.l_num2,
                                                    #.l_num3)
                                               wvr c_minimum
                                               @.l_num1 #.g_num1
                                               @.l_num2 #.g_num2
                                               @.l_num3 #.g_num3
                                               where
                                                   dimension l_num1, l_num2, l_num3;
                                                   c_minimum = #.l_num1 >= 0 and
                                                               #.l_num2 >= 0 and
                                                               #.l_num3 >= 0;
                                               end;
                    end;
end
```

- **XMLCSTranslator** is the concrete translator for generating a custom XML representation of a given *LayeredCompositeService* object. It implements the *CompositeServiceTranslator* interface and defines the *generateFormalLangCode* method. Once the translation is complete, the method returns the complete name (with extension) and location of the XML file to which the translation is written. In case any failure occurs during the translation process, *null* is returned by the method and relevant error messages are recorded in the log file associated with the given logger object.

Similar to the Lucid translation, the XML representation is also written to a file with the translated composite service's name as its title, although the name is preceded by "CSXML_" and the file extension is *.xml*. This file too is stored in the destination location obtained from the given *CSConfiguration* object.

The format of the XML translation, i.e., the XML elements and their arrangement within the resultant file, is exactly the same as that of the composite service XML file repository described in Section 4.3.3. The only difference is that while the repository may hold more than one composite service descriptions, the translation will always describe exactly one composite service, and, therefore, the translation will always have only one "compositeservice" sub-element under the root "compositeservices" element. As an example, consider Listing 4.2, depicting the XML representation of the Range composite service illustrated in Figure 17.

The purpose of having this translator module as part of our framework is to be able to generate a simple and clear, albeit custom, human-readable representation of a *LayeredCompositeService* object in order to aid better understanding of the service's structure. For a more standardized solution, a translator to an extended form of WS-BPEL capable of representing all the features of a layered composite service could be designed and plugged into the framework.

Listing 4.2: XML Translation of Range Composite Service

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<compositeservices>
    <compositeservice>
        <csname value="range"/>
        <csinputs>
            <instance name="int : num1"/>
            <instance name="int : num2"/>
            <instance name="int : num3"/>
        </csinputs>
        <csoutputs>
            <instance name="int : diff"/>
        </csoutputs>
        <cseffects>
        <instance name="int : max"/>
        <instance name="int : min"/>
        <instance name="int : diff"/>
    </cseffects>
    <csconstraints>
        <instance>
            <servicename name="maximum"/>
            <literalvalue name="0"/>
            <type name="num1"/>
            <operator name="&gt;="/>
        </instance>
        <instance>
            <servicename name="maximum"/>
            <literalvalue name="0"/>
            <type name="num2"/>
            <operator name="&gt;="/>
        </instance>
        <instance>
            <servicename name="maximum"/>
            <literalvalue name="0"/>
            <type name="num3"/>
            <operator name="&gt;="/>
        </instance>
        <instance>
            <servicename name="minimum"/>
            <literalvalue name="0"/>
            <type name="num1"/>
            <operator name="&gt;="/>
        </instance>
        <instance>
            <servicename name="minimum"/>
            <literalvalue name="0"/>
            <type name="num2"/>
            <operator name="&gt;="/>
        </instance>
        <instance>
            <servicename name="minimum"/>
            <literalvalue name="0"/>
            <type name="num3"/>
            <operator name="&gt;="/>
        </instance>
    </csconstraints>
    <csplan>
        <servicelayer index="0">
            <servicenode>
                <service name="maximum"/>
                <constraints>
                <instance>
                    <servicename name=‘‘maximum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num1’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘maximum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num2’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘maximum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num3’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
            </constraints>
            <predecessors/>
        </servicenode>
        <servicenode>
            <service name=‘‘minimum’’/>
            <constraints>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num1’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num2’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num3’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
            </constraints>
            <predecessors/>
        </servicenode>
    </servicelayer>
    <servicelayer  index=‘‘1’’>
```

```
        <servicenode>
            <service name=‘‘difference’’/>
            <constraints/>
            <predecessors>
                <instance layerindex=‘‘0’’ name=‘‘maximum’’/>
                <instance layerindex=‘‘0’’ name=‘‘minimum’’/>
            </predecessors>
        </servicenode>
    </servicelayer>
</csplan>
<csatomicservices>
    <service name=‘‘maximum’’>
        <inputs>
            <instance name=‘‘int : num1’’/>
            <instance name=‘‘int : num2’’/>
            <instance name=‘‘int : num3’’/>
        </inputs>
        <outputs>
            <instance name=‘‘int : max’’/>
        </outputs>
        <constraints>
            <instance>
                <servicename name=‘‘maximum’’/>
                <literalvalue name=‘‘0’’/>
                <type name=‘‘num1’’/>
                <operator name=‘‘&gt;=’’/>
            </instance>
            <instance>
                <servicename name=‘‘maximum’’/>
                <literalvalue name=‘‘0’’/>
                <type name=‘‘num2’’/>
                <operator name=‘‘&gt;=’’/>
            </instance>
            <instance>
                <servicename name=‘‘maximum’’/>
                <literalvalue name=‘‘0’’/>
                <type name=‘‘num3’’/>
                <operator name=‘‘&gt;=’’/>
            </instance>
        </constraints>
        <effects>
            <instance name=‘‘int : max’’/>
        </effects>
    </service>
    <service name=‘‘minimum’’>
        <inputs>
            <instance name=‘‘int : num1’’/>
```

```
                <instance name=‘‘int : num2’’/>
                <instance name=‘‘int : num3’’/>
            </inputs>
            <outputs>
                <instance name=‘‘int : min’’/>
            </outputs>
            <constraints>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num1’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num2’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
                <instance>
                    <servicename name=‘‘minimum’’/>
                    < literalvalue  name=‘‘0’’/>
                    <type name=‘‘num3’’/>
                    <operator name=‘‘&gt;=’’/>
                </instance>
            </constraints>
            <effects>
                <instance name=‘‘int : min’’/>
            </effects>
        </service>
        <service name=‘‘difference’’>
            <inputs>
                <instance name=‘‘int : max’’/>
                <instance name=‘‘int : min’’/>
            </inputs>
            <outputs>
                <instance name=‘‘int : diff ’’/>
            </outputs>
            <constraints/>
            <effects>
                <instance name=‘‘int : diff ’’/>
            </effects>
        </service>
    </csatomicservices>
  </compositeservice>
</compositeservices>
```

- **DotGraphCSTranslator** is the concrete translator for generating a DOT graph representation of a given *LayeredCompositeService* object. It implements the *CompositeServiceTranslator* interface and defines the *generateFormalLangCode* method, making it responsible for performing the following tasks:
  - Translating the composite service extracted from the *CSConfiguration* object supplied as input to the *generateFormalLangCode* method into a DOT program, which, when executed, would generate a graphical representation of the composite service.
  - Writing the generated program into a *.dot* file with the composite service's name preceded by "CSDot_" as its title and storing it in the destination location obtained from the given *CSConfiguration* object.
  - Using the DOT executable file details from the composite service configuration for executing the generated DOT program to produce a *.png* image file (with the same name and location as the source DOT program file) containing the graphical representation of the composite service. For successful translations, the *generateFormalLangCode* method terminates by returning the complete name (with extension) and location of the DOT program file. In case any failure occurs during the translation process, *null* is returned by the method and relevant error messages are recorded in the log file associated with the given logger object.

The purpose of having this translator module as part of our framework is to aid visualization of composite services, particularly those with large and complex structures, with which the user might be completely unfamiliar. While our XML representation is concerned with providing detailed information about a composite service and its structure, our DOT graph equivalent serves to enhance the understanding of that information through visual diagrammatic means.

As an example, consider Figure 18, depicting the DOT graph representation of the Range composite service illustrated in Figure 17. Each service-node belonging to the constraint-aware plan of the composition is depicted as a different colored rectangle containing a diamond (labeled as 'C') representing the node's constraints and a circle (labeled as 'W') representing the node's web service. Each of these nodes is enclosed in a gray-colored rectangle representing the service layer to which it belongs. The inputs accepted and outputs generated by each component service are depicted through directed arcs labeled with the parameter names and data types. Finally, the *Output Accumulator* unit illustrated in gray in the graph is the node responsible for forming a collection of those output parameters generated by component services that are expected as output from the composite service as a whole (explained in Section 2.1.4 and discussed further in Section 4.2).

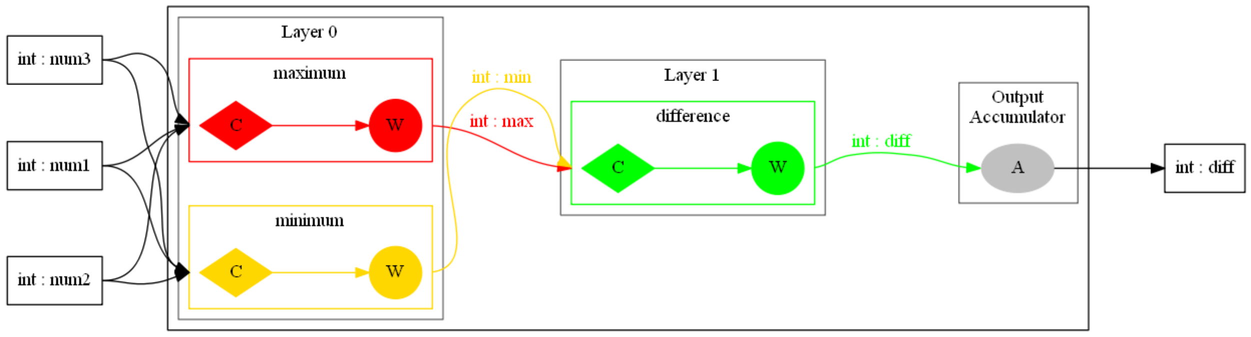


Figure 18: DOT Translation of Range Composite Service

Translators to other target languages/models (depicted as *OtherCSTranslator* in Figure 16) can also be easily added to the existing architecture in future by implementing the *CompositeServiceTranslator* interface and defining the *generateFormalLangCode* method to do the language-specific translation.

4.2 Composite Service to OBJECTIVE LUCID Translation

The implementation and operation of our module for translating constraint-aware composite services (generated by the service composition mechanism discussed in Chapter 3) into their equivalent OBJECTIVE LUCID programs capable of being simulated/executed on GIPSY for

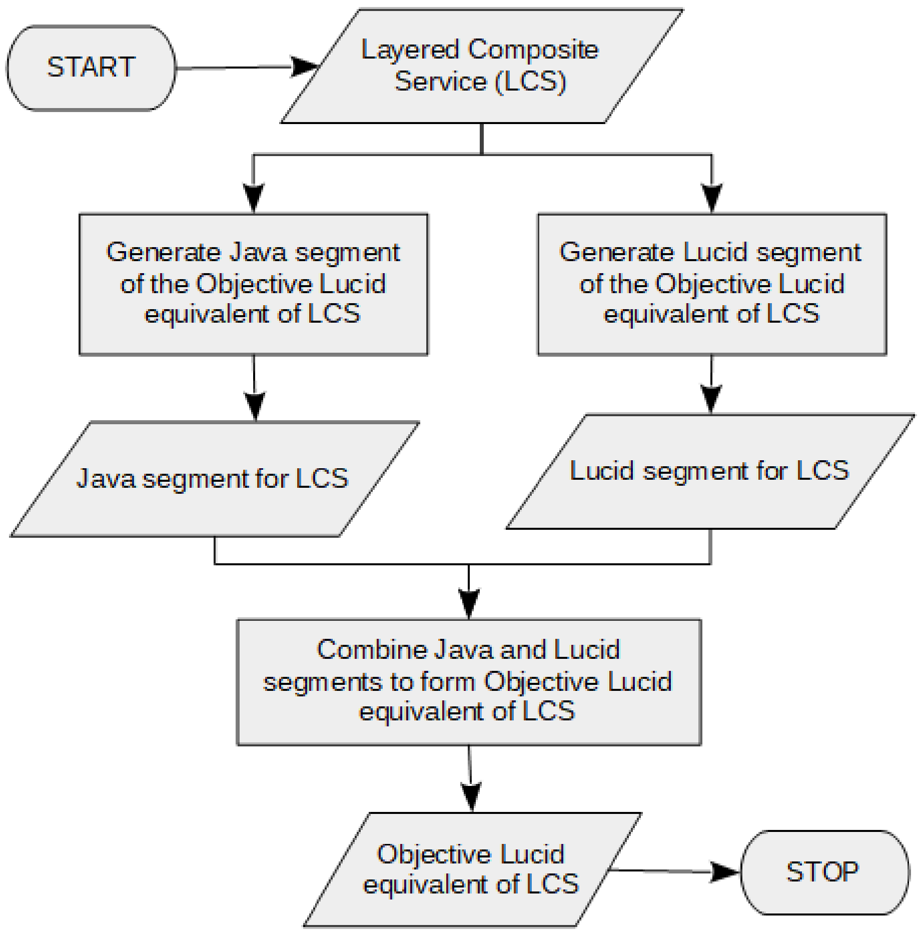

START
Layered Composite Service (LCS)
Generate Java segment of the Objective Lucid equivalent of LCS
Generate Lucid segment of the Objective Lucid equivalent of LCS
Java segment for LCS
Lucid segment for LCS
Combine Java and Lucid segments to form Objective Lucid equivalent of LCS
Objective Lucid equivalent of LCS
STOP

Figure 19: Composite Service to OBJECTIVE LUCID Translation Methodology

verification purposes is based on a set of algorithms that together define our translation methodology (depicted in Figure 19). In this section, we present and explain all these constituent algorithms using the Range composite service depicted in Figure 17 and its OBJECTIVE LUCID translation shown in Listing 4.1 as an example in order to facilitate a clear understanding of our translation process.

Algorithm 7 drives the translation process, invoking the other algorithms involved in the procedure as and when required. It consists of four major steps: (1) *ValidateInpValues* (line 1) responsible for ensuring that, for each input required by the given composite service, a value that matches the respective input's data type is available, failing which the translation process is immediately aborted, (2) *GenerateJavaSegment* (line 2) for producing the JAVA

Algorithm 7 TranslateCSToOLucid

Input: *CS* (composite service), *inputs* (set of CS input names, data types and values)
Output: *oLucidProg* (OBJECTIVE LUCID program), or $\emptyset$ (in case of failure)

```
1: if (ValidateInpValues(inputs)) then
2:     javaSeg = GenerateJavaSegment(CS)
3:     oLucidSeg = GenerateOLucidSegment(CS, inputs)
4:     oLucidProg = javaSeg ∪ oLucidSeg
5:     return oLucidProg
6: else
7:     return ∅
8: end if
```

segment (e.g., Listing 4.1, lines 1 - 92) of the composite service's OBJECTIVE LUCID representation, (3) *GenerateOLucidSegment* (line 3) tasked with generating the OBJECTIVE LUCID segment (e.g., Listing 4.1, lines 94 - 135) of the resultant LUCID program and (4) appending the generated JAVA segment with the OBJECTIVE LUCID segment for constituting the complete OBJECTIVE LUCID translation of the given composite service (lines 4 - 5). As already mentioned, steps 2 - 4 of this algorithm are performed only if all the given input values are successfully validated by *ValidateInpValues*, otherwise the algorithm results in failure (line 7).

Algorithm 8 is responsible for generating the JAVA segment of the OBJECTIVE LUCID translation of a layered composite service. This segment, beginning with a *#JAVA* tag, comprises of a collection of JAVA class and method definitions representing two types of nodes: the output accumulator node and all the component service nodes (depicted in Figure 18). As explained in Section 2.1.4, in LUCID, a composite service is represented as an expression, which can, in a given context, evaluate to only a single value, thereby implying that a composite service, if represented in LUCID, could produce only one output (with potentially different values in different contexts), which would form a highly restrictive service model. However, OBJECTIVE LUCID offers a solution to this restriction in the form of JAVA objects, allowing multiple composite service outputs to be assembled together as data members of a single object, which can then be returned as the computed value of the

Algorithm 8 GenerateJavaSegment

```
Input: $CS$ (composite service)
Output: $javaSeg$ (JAVA segment of CS translation to OBJECTIVE LUCID)
 1: $accmrDataMembs = CS.O$
 2: $accmrCtorParams = CS.O$
 3: $accmrCtorBody$ = initializing $accmrDataMembs$ with respective $accmrCtorParams$
 4: $accmrCtorDef = DefineCtor(accmrCtorParams, accmrCtorBody)$
 5: $accmrClassDef = DefineClass(accmrDataMembs, accmrCtorDef)$
 6: for (each $serviceNode \in CS.plan$) do
 7:     $atomSvcDef = GenerateAtomSvcJavaDef(serviceNode)$
 8:     $atomSvcDefs = atomSvcDefs \cup atomSvcDef$
 9: end for
10: $javaSeg = accmrClassDef \cup atomSvcDefs$
11: return $javaSeg$
```

composite service expression. This task of accumulating composite service outputs in order to construct a JAVA object is performed by the output accumulator node represented as JAVA class *CAWSReqComp* (Listing 4.1, lines 2 - 10, where *CAWS* stands for *Constraint-Aware Web Service*) in the resultant OBJECTIVE LUCID program. The composite service output parameters serve as the private data members of this class. Once all the component services complete their processing, values for each of these parameters are passed as arguments while calling the accumulator class constructor from the OBJECTIVE LUCID segment (Listing 4.1, line 99). These arguments are then used within the *CAWSReqComp* constructor to initialize the data members, i.e., the composite service outputs, thereby constructing the intended JAVA object.

Lines 1 - 5 of Algorithm 8 are dedicated to stepwise generating the JAVA class definition of the output accumulator node, beginning with defining the composite service outputs as its data members (line 1) and constructor parameters (line 2), creating initialization statements that form the constructor's body (line 3), using the constructed parameter list and body to build the complete constructor definition (line 4) and, finally, using the data members and constructor to build the *CAWSReqComp* class definition (line 5). Lines 6 - 9 of the algorithm are responsible for generating the entire collection of JAVA class and method definitions representing the component services that together constitute the given composite service CS. For each service-node belonging to the composite service's constraint-aware plan (line 6), *GenerateAtomSvcJavaDef* (Algorithm 9) constructs a pair of JAVA class and method definitions – *atomSvcDef* (line 7), which is then appended to the set of definitions generated until that point (line 8). Once all component service definitions have been produced, construction of the resultant program's JAVA segment is completed by appending the output accumulator definition with the component service definition collection (line 10).

Algorithm 9 is invoked by Algorithm 8 (line 7) for generating the pair of JAVA class and free function definitions that together define and provide the means of triggering the operation of a component service. While the JAVA class is tasked with processing the inputs provided to the component service in order to produce the desired outputs and assembling them into a single unit as data members, the method is responsible for creating an object of this class by supplying the inputs required by the service, triggering its processing using the object created and, finally, returning the object updated with the outputs obtained to the OBJECTIVE LUCID segment from where this method is called. As an example, consider the call to the *difference* method in Listing 4.1 (line 106). This is the free function that forms part of the JAVA definition of the *difference* component service (lines 12 - 36). The outputs produced by the *maximum* and *minimum* services, which serve as inputs to the *difference* service, are supplied as arguments to this function call to be used by the function as arguments while calling the corresponding service class (*CAWSdifference*) constructor. This constructor then uses these arguments for initializing the service's input parameters acting as its class data members. The output data members, in the meantime, are initialized with dummy values based on their data type: 0 for *int*, 0.0 for *double*, *false* for *boolean* and a whitespace character for *char* and *String*. Once the service object *oCAWSdifference* is created, the *difference* function uses it to invoke the *process* member function of the service class, which is responsible for processing the given inputs and updating the output data members with the results obtained, following which the *difference* function returns the

Algorithm 9 GenerateAtomSvcJavaDef

Input: *serviceNode* (service-node from composite service constraint-aware plan)
Output: *atomSvcDef* (Java code representing *serviceNode*)

```
 1: $inpDataMembs = serviceNode.service.I$
 2: $outpDataMembs = serviceNode.service.O$
 3: $svcDataMembs = inpDataMembs \cup outpDataMembs$

 4: $svcCtorParams = serviceNode.service.I$
 5: $inpInitStmts$ = initializing $inpDataMembs$ with respective $svcCtorParams$
 6: $outpInitStmts$ = initializing $outpDataMembs$ based on data type
 7: $svcCtorBody = inpInitStmts \cup outpInitStmts$
 8: $svcCtorDef = DefineCtor(svcCtorParams, svcCtorBody)$

 9: $procFnBody$ = operations performed by $serviceNode.service$
10: $procFnDef = DefineFunc(\emptyset, \emptyset, procFnBody)$

11: $svcMembFns = svcCtorDef \cup procFnDef$
12: $svcClassDef = DefineClass(svcDataMembs, svcMembFns)$

13: $freeFnParams = serviceNode.service.I$
14: $freeFnBody$ = creating $svcClass$ object and using it to invoke $procFn$
15: $freeFnDef = DefineFunc(svcClass, freeFnParams, freeFnBody)$

16: $atomSvcDef = svcClassDef \cup freeFnDef$
17: return $atomSvcDef$
```

updated *oCAWSdifference* object and, hence, the service outputs to the Objective Lucid segment.

It should be noted here that the *process* method in a service's Java class is a container for holding the simulated or actual implementation of or a link to a real online component service. However, as part of our current implementation, its body comprises of nothing but simple placeholder statements that assign dummy values (different from those used by the constructor) to output data members based on their data type: 10 for *int*, 20.0 for *double*, *true* for *boolean*, *'a'* for *char* and "*test*" for *String*. The purpose of these statements, at this point, is to indicate if each component service gets triggered correctly during execution on GIPSY (provided all its constraints are met) and whether a final output compliant with the expected data type is obtained, i.e., to ensure that our current solution passes the basic sanity checks. For more advanced testing and verification, as part of future extensions, we plan to include an additional component in the atomic service model described by Laleh et al. (Definition 1) that would specify the implementation (whether simulated, actual or linked) of the service and could be extracted by the OBJECTIVE LUCID translator module so as to replace the current placeholder implementation of the corresponding service's *process* method.

For generating the currently presented JAVA class and method definitions of a component service, Algorithm 9 uses a stepwise approach similar to the one employed by Algorithm 8. While lines 1 - 3 of the algorithm define the service's input and output parameters as its class data members, lines 4 - 8 create a constructor for this class, using a parameter list composed of the service's inputs (line 4) and a body consisting of data member initialization statements (lines 5 - 7). The *process* member function of this class is defined by lines 9 - 10 of the algorithm, where its *void* return type and empty parameter list are indicated by using $\emptyset$ as the first two arguments to the *DefineFunc* procedure. All these elements generated – data members, constructor and *process* method – are then used for building the component service's class definition (lines 11 - 12). Definition of the component service's free function is also constructed in a similar fashion, using the same *DefineFunc* procedure (line 15) that assembles the *process* method's elements. The difference here, however, is that the free function accepts its corresponding service's inputs as parameters (line 13), uses them to create an object of the service class, invokes the *process* member function through it (line 14), and, finally, returns the object created (indicated using return type *svcClass* as first argument to *DefineFunc* on line 15) to the OBJECTIVE LUCID segment. When appended to the service class definition, this method completes the JAVA representation of a component service in the OBJECTIVE LUCID translation of its corresponding composite service.

Algorithm 10, similar to Algorithms 8 and 9, incrementally builds the LUCID segment of the OBJECTIVE LUCID translation of a layered composite service, sequentially generating its various logical sections and finally assembling them together for producing the complete segment. As explained in Sections 2.1.3 and 2.1.4, the inputs of a composite service also

Algorithm 10 GenerateOLucidSegment

Input: *CS* (composite service), *inputs* (set of CS input names, data types and values)
Output: *oLucidSeg* (LUCID segment of CS translation to OBJECTIVE LUCID)

```
 1: $globalDims = CS.I$
 2: $globalContext$ = set of respective $CS.I : inputs.value$ pairs
 3: $mainExpr$ = single-variable expression evaluated in $globalContext$

 4: $accmrInps = accmrDims = CS.O$
 5: for (each $serviceNode \in CS.plan$) do
 6:   $csOutps = serviceNode.service.O \cap CS.O$
 7:   for (each $output \in csOutps$) do
 8:     $accmrContext = accmrContext \cup output : oSvcClass.output$
 9:   end for
10: end for
11: $accmrConstrs = \emptyset$

12: for (each $serviceNode \in CS.plan$) do
13:   $atomSvcDef = GenerateAtomSvcLucidDef(serviceNode, globalDims)$
14:   $atomSvcDefs = atomSvcDefs \cup atomSvcDef$
15: end for

16: $accmrDef = DefineSvc(\text{“CAWSReqComp”}, accmrInps, accmrDims, accmrContext,$
    $accmrConstrs, atomSvcDefs)$

17: $oLucidSeg = mainExpr \cup globalDims \cup accmrDef$
18: return $oLucidSeg$
```

act as its contextual dimensions. Therefore, in order to generate the main LUCID expression representing the outcome of the composite service, line 1 of the algorithm constructs a global dimension list comprising of the composite service input parameters, which are then paired with their respective values provided by the user (through *inputs*) for defining the context in which the main expression needs to be evaluated (lines 2 - 3).

For constructing the LUCID expression and associated *where* clause that together represent the output accumulator node of the composition, the algorithm uses the *DefineSvc* procedure (line 16), which is responsible for building a service-node's LUCID representation once all its elements – service name, service inputs, service dimensions, service evaluation context, service constraints and other component service definitions that may influence its outcome – have been determined. Since the output accumulator node performs the task of collecting the component service outputs that together constitute the outputs expected from the composite service as a unit, its inputs and dimensions are the same as the composite service output parameters (line 4). In order to define the accumulator's evaluation context, we need to pair its contextual dimensions with their respective values generated by the various component services. Therefore, lines 5 - 10 of the algorithm iterate through the entire constraint-aware plan of the composition, determining which component service generates which composite service output(s) (lines 6 - 9) and pairing each accumulator dimension with the matched output parameter, i.e., the appropriate output data member of the component service's Java class object (line 8). Since, in this thesis, we do not take requester or external constraints into consideration (discussed in Section 1.2), the composite service and, hence, the output accumulator node do not have any separate constraints placed on them (line 11); the only constraints that apply to the composite service are the internal constraints imposed on its component services by their providers. Lucid representation of each of these component services, which also forms part of the accumulator node's *where* clause, is generated by Algorithm 11 invoked iteratively from Algorithm 10 (lines 12 - 15) for each service-node in the composition's constraint-aware plan. Once all these constituent definitions get generated, the *DefineSvc* procedure organizes them into the Lucid representation of the accumulator node, using its corresponding Java class name (*CAWSReqComp*) for producing the statement that makes a call to its constructor from the Lucid segment (line 16). Finally, the generated accumulator definition together with the main expression and global dimension list constructed earlier in the algorithm are assembled to produce the complete Objective Lucid segment of the translation (line 17).

Algorithm 11 is iteratively invoked by Algorithm 10 (line 13) for generating the Objective Lucid representation of each component service belonging to a composition. This algorithm follows the same approach for building these representations as adopted by its preceding algorithm for constructing the output accumulator node representation. 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 (lines 1 - 4). As explained in Section 2.1.3, computation of a LUCID expression is triggered only after all the constraints placed on it get evaluated to *true*, which, in turn, requires values of the constraint-features to be computed and compared with their corresponding literal values. As discussed in Section 3.2, optimization of constraint verification points within a constraint-aware plan can result in a component service-node being imposed with constraints placed on the inputs of its enclosed service itself and/or those transferred to it from other nodes in the plan during constraint adjustment. Since, in order to compute a component service expression in LUCID, all its attached constraints – whether applicable on its own inputs or on those of other component services – must be evaluated, it becomes necessary to include these additional constraint-features (if any) in the component service's dimension list as well as its evaluation context, which specifies the source of their values (whether user or other component service) – required for their computation.

Once the contextual dimensions for the given component service's LUCID expression have been defined, the algorithm proceeds to building its evaluation context, pairing its dimensions with values either received from the user or generated as outputs by other component services (lines 5 - 15). Any dimension that is found to match with an output parameter of a predecessor of the given service (line 6) is paired with the appropriate data member of the predecessor service's JAVA class object (line 9). Since the validation checks and pruning activities performed during the construction of a composite service (in Algorithm 5) ensure that all the inputs of component services are available within the composition, it can be concluded that those dimensions that do not receive their values from another component must be receiving them from the user as part of the composition request. Therefore, the remaining unpaired local dimensions of the given component service (line 12) are paired with their global counterparts (lines 13 - 15) to which the user-provided values are assigned in Algorithm 10 during global context definition (line 2), thereby completing the component service's local context definition. Finally, the inputs and constraints attached to the given service-node are extracted (lines 16 - 17) to be used as the second and fifth arguments

Algorithm 11 GenerateAtomSvcLucidDef

```
Input: $serviceNode$ (service-node from composite service constraint-aware plan),
       $globalDims$ (set of composite service's global dimensions)
Output: $atomSvcDef$ (OBJECTIVE LUCID code representing $serviceNode$)
 1: for (each $constraint \in serviceNode.C$) do
 2:     $cnstrFeatures = cnstrFeatures \cup constraint.feature$
 3: end for
 4: $localDims = serviceNode.service.I \cup cnstrFeatures$

 5: for (each $predNode \in serviceNode.predecessors$) do
 6:     $paramsFromCurrPred = predNode.service.O \cap localDims$
 7:     $paramsFromPreds = paramsFromPreds \cup paramsFromCurrPred$
 8:     for (each $parameter \in paramsFromCurrPred$) do
 9:         $localContext$ = set of respective $parameter : oPredSvcClass.parameter$
10:     end for
11: end for
12: $paramsFromUser = localDims - paramsFromPreds$
13: for (each $parameter \in paramsFromUser$) do
14:     $localContext = localContext \cup$ set of respective $parameter : globalDims.parameter$
15: end for

16: $svcInps = serviceNode.service.I$
17: $svcCnstrs = serviceNode.C$
18: $atomSvcDef = DefineSvc(serviceNode.service.name, svcInps, localDims,$
    $localContext, svcCnstrs, \emptyset)$
19: return $atomSvcDef$
```

respectively to the *DefineSvc* procedure (line 18) while its last argument is left blank (depicted as $\emptyset$) since there are no component service definitions to be included in the given service's *where* clause. Using these arguments together with the local dimension list and evaluation context defined earlier in addition to the component service name required for producing the LUCID expression that makes a call to its corresponding free function in the JAVA segment, *DefineSvc* builds the complete OBJECTIVE LUCID representation of the given component service.

4.3 Additional Features

In order to add more flexibility to our service translator framework and to enable its use as a tool/application, we introduce some additional functionalities similar to those added to the service composition implementation (described in Section 3.5.4) into it. In this section, we discuss these additional features and the related architecture.

4.3.1 Service Translation Driver

The service translation driver, similar to the composition driver (discussed in Section 3.5.4.1), is responsible for prompting the user to provide the inputs required for executing the service translation process, for triggering the various phases involved in the process in the proper sequence and for displaying the final status (success/failure) of the process on the console. More specifically, the driver performs the following tasks:

1. Prompt the user on the console to select a mode of input for providing details of the composite service configuration. At present, the user can select from console and XML file modes, although the architecture in place allows these options to be extended to other modes as well (discussed in Section 4.3.2).

2. Create a logger object (described in Section 3.5.4.3), which would be passed across methods to allow error messages generated throughout the process to be recorded in a log file.

3. If XML file is selected as the mode of input in step 1, prompt the user to provide XML configuration file path.

4. Depending on the selected mode of input, invoke the corresponding composite service configuration reader, passing it the logger object created, to fetch the composite service to be translated, the target language of the translation and other necessary information from the user and use it to create a composite service configuration object (described in Section 4.3.2).

5. If the configuration building process fails at any point, display a failure message and invite the user to check the log file for error details. However, if a valid configuration is successfully constructed, depending on the target language indicated by the user in the configuration, trigger the translation process, passing it the configuration and logger objects created. At present, the user can select from Lucid (more specifically, Objective Lucid), XML and DOT languages, although the architecture in place allows these options to be extended to other target languages as well (discussed Section 4.1).

6. If the translation process is successful, invite the user to check the file to which the translation has been written. However, if the translation process fails at any point, display a failure message and invite the user to check the log file for error details.

For example, Listing 4.1, Listing 4.2 and Figure 18 respectively show the contents of the Objective Lucid (*CSLucid_range.ipl*), XML (*CSXML_range.xml*) and DOT (*CSDot_range.png*) translation files generated for the Range composite service depicted in Figure 17. In an alternate scenario, if the user-specified repository from which the composite service to be translated needs to be extracted does not exhibit an acceptable file format, a log file same as the one depicted in Listing 4.3 gets generated instead of a translation file.

Listing 4.3: Invalid Repository File Type Log (in Text File Format)

```
Invalid repository file type in the given composite service configuration.
Only serialized Java object file or XML file can be parsed.
```

4.3.2 Composite Service Configuration and Readers

A *CSConfiguration* object contains all the information provided by the user that is required to execute the service translation process. It consists of the following elements:

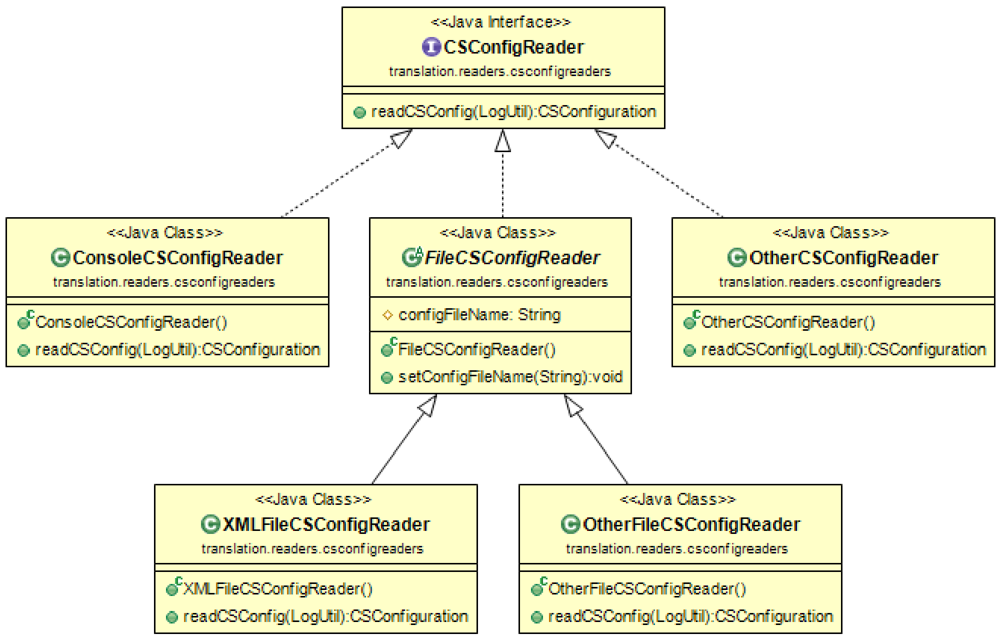
- **Composite Service:** The layered composite service object to be translated (discussed in Section 4.3.3).

- **Composite Service Input Details:** A list of records, each of which is composed of the name, data type and value of a specific input parameter of the composite service to be translated.
- **Target Language:** The language to which the composite service needs to be translated. At present, our implementation supports OBJECTIVE LUCID, XML and DOT languages (discussed in Section 4.1).
- **Destination Folder:** Complete path of the folder where the translation file is placed once generated.
- **DOT Executable Name:** Complete path of the DOT executable file (*dot.exe*) required for executing the DOT program generated by the DOT translator module in order to produce a *.png* image file containing the graphical representation of the translated composite service.

Once a *CSConfiguration* object is created, its target language component is used to determine the appropriate translator module to be invoked, which accepts the configuration object as an argument. While the XML translator module makes use of only the composite service object contained within the configuration object for generating its target language representation, the OBJECTIVE LUCID and DOT translators also require the input details and the DOT executable file location respectively in order to complete the translation. The destination folder component of the configuration is used by all of the currently available translator modules as the location for placing the translation files that each of them generates.

As mentioned in Section 4.3.1, a user can opt for different modes for supplying the composite service configuration details. We have designed and implemented an architecture (as depicted in Figure 20) that allows modular addition and removal of readers for each of these modes. At present, this implementation can support console and XML file readers. Major design and implementation specifications of this architecture have been listed below:

- **CSConfigReader** is the interface to be implemented by all concrete composite service configuration readers. It declares the *readCSConfig* method, which should be defined



<<Java Interface>>
CSConfigReader
translation.readers.csconfigreaders
readCSConfig(LogUtil):CSConfiguration
<<Java Class>>
ConsoleCSConfigReader
translation.readers.csconfigreaders
ConsoleCSConfigReader()
readCSConfig(LogUtil):CSConfiguration
<<Java Class>>
FileCSConfigReader
translation.readers.csconfigreaders
configFileName: String
FileCSConfigReader()
setConfigFileName(String):void
<<Java Class>>
OtherCSConfigReader
translation.readers.csconfigreaders
OtherCSConfigReader()
readCSConfig(LogUtil):CSConfiguration
<<Java Class>>
XMLFileCSConfigReader
translation.readers.csconfigreaders
XMLFileCSConfigReader()
readCSConfig(LogUtil):CSConfiguration
<<Java Class>>
OtherFileCSConfigReader
translation.readers.csconfigreaders
OtherFileCSConfigReader()
readCSConfig(LogUtil):CSConfiguration

Figure 20: Composite Service Configuration Reader Architecture

to accept composite service configuration details from the user based on the mode of input being handled by each concrete reader.

The logger object created by the translation driver (described in Section 4.3.1) is provided as an argument to the *readCSConfig* method in order to allow any error messages generated during the construction of the *CSConfiguration* object to be recorded in a log file.

- **ConsoleCSConfigReader** is the concrete composite service configuration reader for interacting with the user through the console to obtain composite service configuration details. It implements the *CSConfigReader* interface and defines the *readCSConfig* method. After reading and processing all the required information, the method creates a *CSConfiguration* object to be used for service translation.

- **FileCSConfigReader** is the abstract class to be extended by all concrete composite service configuration file readers. It implements the *CSConfigReader* interface but does not define the *readCSConfig* method; the method is expected to be defined by the concrete file readers.

This class contains a *configFileName* data member and a mutator method for assigning a value to it. The *configFileName* member is inherited by all concrete file readers and stores the complete path of the file to be read by them.

- **XMLFileCSConfigReader** is the concrete composite service configuration reader for extracting composite service configuration details from a user-specified XML file. It extends the *FileCSConfigReader* class and defines the *readCSConfig* method. After reading and processing all the required information, the method creates a *CSConfiguration* object to be used for service translation.

Readers for other file formats (depicted as *OtherFileCSConfigReader* in Figure 20) can be easily added to the existing architecture by extending the *FileCSConfigReader* class and defining the *readCSConfig* method to do the file-specific parsing. To include readers for other modes of input, such as, databases, a new class (depicted as *OtherCSConfigReader* in Figure 20) can be added to this hierarchy and made to implement the *CSConfigReader* interface while defining the required behavior in the *readCSConfig* method.

The *readCSConfig* method for the console reader is responsible for performing the following tasks:

1. Consecutively prompting the user on the console to provide the value for each element of a composite service configuration according to the following rules:

    - **CS Repository Filename** must include the complete name (with extension) and path of the repository file containing description of the composite service to be translated. For now, only *.txt* and *.xml* are recognized as acceptable file extensions.

    - **Composite Service Name** must be the name of the composite service to be extracted from the given repository for translation.

    - **Target Language Name** must be the name of the formal language to which the specified composite service needs to be translated. For now, only Lucid, XML and DOT are recognized as acceptable target languages.

- **CS Input Values** must be in accordance with the name and data type mentioned in the prompt for each specific composite service input. User will be prompted for input values only if the chosen target language is Lucid.
- **DOT Executable Filename** must include the complete name (with extension) and path of the DOT executable file (*dot.exe*). User will be prompted for this file location only if the chosen target language is DOT.

2. Depending on the type of the specified repository, invoking the corresponding composite service reader (discussed in Section 4.3.3) in order to fetch the specified service's description from the repository and using it to create the *LayeredCompositeService* object to be translated into the target language.

3. If the chosen target language is Lucid, creating a list of input records using the input name and data type from the *LayeredCompositeService* object and their respective values received from the user.

4. Creating a *CSConfiguration* object using the specified repository file location as the destination folder for the translation file to be generated together with all the other information collected in the preceding steps.

5. In case of any failures during the creation of the *CSConfiguration* object, recording proper error messages in the log file associated with the logger object accepted as an argument and aborting the translation process immediately.

The *readCSConfig* method for the XML file reader performs the same tasks as those performed by its console counterpart with the only distinction being that, instead of reading the configuration details from the console, this method obtains them by parsing a user-specified XML configuration file, which comprises of the following elements:

- **csconfig:** It is the root element of the XML file.
- **csrepofilename:** It appears once in the XML file, as a sub-element of the root. The

value assigned to its "value" attribute is the complete name (with extension) and path of the composite service repository file.

- **csname:** It appears once in the XML file, as a sub-element of the root. The value assigned to its "value" attribute is the name of the composite service to be extracted from the given repository for translation.
- **targetlang:** It appears once in the XML file, as a sub-element of the root. The value assigned to its "value" attribute is the name of the formal language to which the specified composite service needs to be translated.
- **input:** It appears once for each composite service input, as a sub-element of the root. It has three sub-elements called "name", "type" and "value". The values assigned to the "value" attributes of these sub-elements are, respectively, the corresponding input's name, data type and value. This element is required only for translation to LUCID.
- **dotexename:** It appears once in the XML file, as a sub-element of the root. The value assigned to its "value" attribute is the complete name (with extension) and path of the DOT executable file (*dot.exe*). This element is required only for translation to DOT.

The same format and rules apply to composite service configuration's XML elements as defined for the console mode. As examples, consider Listings 4.4, 4.5 and 4.6, depicting the contents of the XML configuration files for translating the Range composite service (depicted in Figure 17) into XML, DOT and OBJECTIVE LUCID respectively.

Listing 4.4: Range CS Configuration for XML Translation (in XML File Format)

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<csconfig>
    <csrepofilename value="testinput/xmltranslatortests/Serialized_Repository.txt"/>
    <csname value="range"/>
    <targetlang value="XML"/>
</csconfig>
```

Listing 4.5: Range CS Configuration for DOT Translation (in XML File Format)

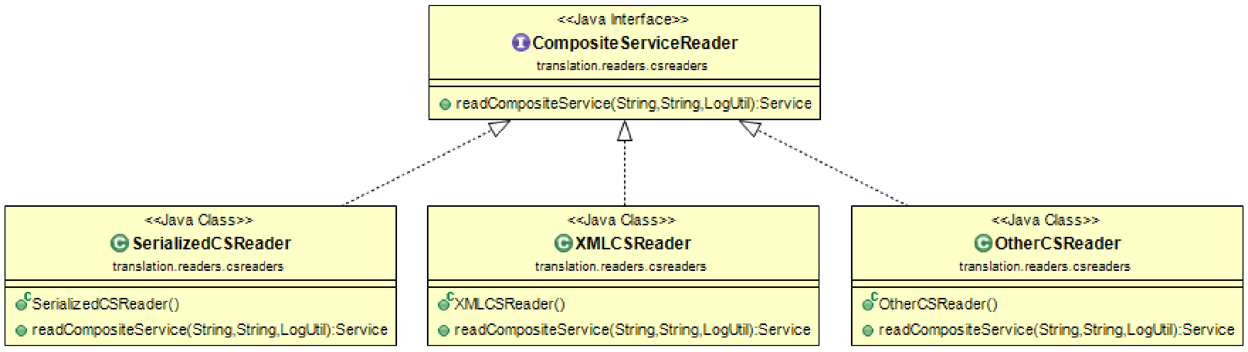
```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<csconfig>
    <csrepofilename value="testinput/dottranslatortests/XML_Repository.xml"/>
    <csname value="range"/>
    <targetlang value="Dot"/>
    <dotexename value="D:\Graphviz\graphviz−2.38\release\bin\dot.exe"/>
</csconfig>
```

Listing 4.6: Range CS Configuration for Lucid Translation (in XML File Format)

```
<?xml version="1.0" encoding="UTF−8" standalone="no"?>
<csconfig>
    <csrepofilename value="testinput/ lucidtranslatortests /Serialized_Repository . txt ''/>
    <csname value="range"/>
    <targetlang value="Lucid"/>
    <input>
        <name value="num1"/>
        <type value="int''/>
        <value value="10"/>
    </input>
    <input>
        <name value="num2"/>
        <type value="int''/>
        <value value="12"/>
    </input>
    <input>
        <name value="num3"/>
        <type value="int''/>
        <value value="14"/>
    </input>
</csconfig>
```

4.3.3 Composite Service Readers

As mentioned in Section 4.3.2, a user can specify different types of composite service repositories as part of composite service configurations. Based on the repository type specified in a configuration, the operating composite service configuration reader invokes a particular composite service reader for parsing the given repository, using the composite service name from the configuration for locating the service in the repository and transforming the service details so obtained into a *LayeredCompositeService* object (as described in Section 3.5.4.6), which is eventually translated into the target language selected by the user. The architecture that we have designed and implemented for the composite service readers (as depicted in Figure 21) allows modular addition and removal of reader modules for each of the possible repository types. At present, this implementation can parse serialized JAVA object and XML file repositories. Major design and implementation specifications of this architecture have been listed below:



<<Java Interface>>
CompositeServiceReader
translation.readers.csreaders
readCompositeService(String,String,LogUtil):Service
<<Java Class>>
SerializedCSReader
translation.readers.csreaders
SerializedCSReader()
readCompositeService(String,String,LogUtil):Service
<<Java Class>>
XMLCSReader
translation.readers.csreaders
XMLCSReader()
readCompositeService(String,String,LogUtil):Service
<<Java Class>>
OtherCSReader
translation.readers.csreaders
OtherCSReader()
readCompositeService(String,String,LogUtil):Service

Figure 21: Composite Service Reader Architecture

- **CompositeServiceReader** is the interface to be implemented by all concrete composite service readers. It declares the *readCompositeService* method, which should be defined to parse and search through a composite service repository based on the repository type being handled by each concrete reader.

- **SerializedCSReader** is the concrete reader for extracting a specific composite service from a *.txt* repository file containing serialized composite service JAVA objects. It implements the *CompositeServiceReader* interface and defines the *readCompositeService* method.

- **XMLCSReader** is the concrete reader for extracting a specific composite service from an XML composite service repository file. It implements the *CompositeServiceReader* interface and defines the *readCompositeService* method.

Readers for other repository formats (depicted as *OtherCSReader* in Figure 21) can also be easily added to the existing architecture by implementing the *CompositeServiceReader* interface and defining the *readCompositeService* method to do the format-specific parsing.

The *readCompositeService* method for the serialized composite service reader has been designed to parse a text file containing a serialized JAVA *ArrayList* of one or more *LayeredCompositeService* objects type-cast to their superclass *Service* (defined in the service repository framework [51]). The method accepts the complete name (with extension) and path of this serialized repository file, the name of the composite service to be translated and the logger object created by the translator driver (described in Section 4.3.1) as arguments and uses them to perform the following tasks:

1. Using the *ServiceSerializedParser* defined in the service repository framework [51] to read the list of services contained within the given repository.

2. Searching the extracted *ArrayList* of *Service* objects for the intended composite service by its name and returning its *Service* object, if found.

3. In case the intended composite service is not found in the given repository, using the logger object to record a suitable error message in its associated log file and returning *null* as a trigger for immediate termination of the translation process.

In contrast, the *readCompositeService* method for the XML composite service reader follows a slightly different approach towards creating the *LayeredCompositeService* object required for translation as described below:

1. This method first fetches a list of all the "compositeservice" XML nodes from the user-specified repository file (described later in this section).

2. The list is then searched for the intended composite service by its name. If found, its corresponding "compositeservice" node is completely parsed to extract the composite service name, inputs, outputs, effects and constraints for creating a simple *ConstrainedService* object (defined in the service repository framework [51]),

which is then decorated with the extracted constraint-aware plan element in order to form the required *LayeredCompositeService* object (as described in Section 3.5.4.6). Meanwhile, the other "compositeservice" nodes in the list are discarded as irrelevant.

3. In case the intended composite service is not found in the given repository, the method uses the given logger object to record a suitable error message in its associated log file and returns *null* as a trigger for immediate termination of the translation process.

Any XML composite service repository file to be parsed successfully by this *readCompositeService* method should comprise of the following elements:

- **compositeservices:** It is the root element of the XML file.
- **compositeservice:** It appears as a sub-element of the root, once for every composite service that resides in the repository. All the information about a composite service is stored within the sub-elements of its corresponding "compositeservice" element.
- **csname:** It appears as a sub-element of every "compositeservice" element. The value assigned to its "value" attribute is the corresponding composite service's name.
- **csinputs, csoutputs, cseffects:** Each of these elements appears as a sub-element of every "compositeservice" element. For each composite service input, output and effect, an "instance" sub-element is added to its corresponding element. The value assigned to the "name" attribute of the "instance" element is "*datatype* : *name*" of the input/output/effect parameter.
- **csconstraints:** It appears as a sub-element of every "compositeservice" element. For each composite service constraint, an "instance" sub-element is added to it, which must comprise of four sub-elements (listed below) with the following values assigned to their respective "name" attributes:
  - **servicename:** Name of the component service on which the constraint is placed.

  - **literalvalue:** Literal value to which the constrained feature's value has to be compared for constraint verification.
  - **type:** Constrained feature, which should be one of the component service input parameters specified as *"featuredatatype : featurename"*.
  - **operator:** Relational operator, which defines the type of comparison between the constrained feature and literal value and could be <, >, =, <= or >=.

- **csplan:** It appears as a sub-element of every "compositeservice" element and describes its constituent service-nodes and the relationships between them. It comprises of the following sub-elements:
  - **servicelayer:** It appears as a sub-element of a "csplan" element, once for each service layer in the plan. The value assigned to its "index" attribute is an integer indicating the index of the layer, starting from 0 and increasing by 1 for each subsequent layer.
  - **servicenode:** It appears as a sub-element of a "servicelayer" element, once for each service-node belonging to that layer.
  - **service:** It appears as a sub-element of each "servicenode" element. The value assigned to its "name" attribute is the name of the component service that it represents.
  - **constraints:** It appears as a sub-element of each "servicenode" element. It describes the constraints attached to the service-node, following the same format as that of the "csconstraints" element discussed above.
  - **predecessors:** It appears as a sub-element of each "servicenode" element. For each predecessor of the given service-node, an "instance" sub-element is added to it. The values assigned to the "name" and "layerindex" attributes of the "instance" element are the predecessor's service name and container layer index respectively.

- **csatomicservices:** It appears as a sub-element of every “compositeservice” element and lists the descriptions of all the component services that together constitute the composite service. It contains the following sub-elements:
  - **service:** It appears as a sub-element of a “csatomicservices” element, once for each component service. The value assigned to its “name” attribute is the component service name.
  - **inputs, outputs, effects, constraints:** Each of these elements appears as a sub-element of each “service” element and follows the same format as that of the “csinputs”, “csoutputs”, “cseffects” and “csconstraints” elements respectively.

If there are no values to be provided for an optional service property, such as, constraints or predecessors, the main element of the property should be included in the XML file without any sub-elements. For instance, for a composite service that is not restricted by any constraints, the opening and closing “csconstraints” tags should be included in the XML file without any “instance” sub-elements between them (as shown in Listing 4.7, lines 25 - 26). Listing 4.7 shows the contents of a sample XML composite service repository file containing three composite services (lines 3 - 11, 12 - 74 and 75 - 77).

Listing 4.7: Sample XML Composite Service Repository

```
1  <?xml version=‘‘1.0’’ encoding=‘‘UTF−8’’ standalone=‘‘no’’?>
2  <compositeservices>
3      <compositeservice>
4          <csname value=‘‘range’’/>
5          <csinputs>
6              <instance name=‘‘int : num1’’/>
7              <instance name=‘‘int : num2’’/>
8              <instance name=‘‘int : num3’’/>
9          </csinputs>
10          ...
11     </compositeservice>
12     <compositeservice>
13         <csname value=‘‘CalcPercent’’/>
14         <csinputs>
15             <instance name=‘‘string : StudentID’’/>
16         </csinputs>
17         <csoutputs>
```

```
            <instance name="float : MarksPercentage"/>
        </csoutputs>
        <cseffects>
            <instance name="float : TotalMarks"/>
            <instance name="int : NumberOfCourses"/>
            <instance name="float : MarksPercentage"/>
        </cseffects>
        <csconstraints>
        </csconstraints>
        <csplan>
            <servicelayer index="0">
                <servicenode>
                    <service name="W8"/>
                    <constraints/>
                    <predecessors/>
                </servicenode>
            </servicelayer>
            <servicelayer index="1">
                <servicenode>
                    <service name="W9"/>
                    <constraints/>
                    <predecessors>
                        <instance name="W8" layerindex="0"/>
                    </predecessors>
                </servicenode>
            </servicelayer>
        </csplan>
        <csatomicservices>
            <service name="W8">
                <inputs>
                    <instance name="string : StudentID"/>
                </inputs>
                <outputs>
                    <instance name="float : TotalMarks"/>
                    <instance name="int : NumberOfCourses"/>
                </outputs>
                <constraints/>
                <effects>
                    <instance name="float : TotalMarks"/>
                    <instance name="int : NumberOfCourses"/>
                </effects>
            </service>
            <service name="W9">
                <inputs>
                    <instance name="float : TotalMarks"/>
                    <instance name="int : NumberOfCourses"/>
                </inputs>
                <outputs>
                    <instance name=“float : MarksPercentage”/>
                </outputs>
                <constraints/>
                <effects>
                    <instance name=“float : MarksPercentage”/>
                </effects>
            </service>
        </csatomicservices>
    </compositeservice>
    <compositeservice>
        ...
    </compositeservice>
</compositeservices>
```

4.4 Summary

In this chapter, we presented our methodology for translating the constraint-aware composite services generated by the composition technique described in Chapter 3 into OBJECTIVE LUCID programs that can be simulated/executed on GIPSY for verification and analysis. Although designed primarily for translation to OBJECTIVE LUCID, our translator framework allows easy addition and removal of modules for other target languages/models as well while also allowing some flexibility in modes of input and user-interaction. This concludes our discussion on the service verification procedure depicted in Figure 3. In the next chapter, we present the strategy that we employ for evaluating our solution, the analysis performed and the results obtained together with the conclusions that can be drawn on the extent to which the solution fulfills its design goals.

Chapter 5

Solution Evaluation

In Chapters 3 and 4, we explain the process of composing constraint- and context-aware services in response to a given composition request and translating them into a variety of useful formats, including OBJECTIVE LUCID programs that can be simulated/executed on GIPSY for verification and testing purposes. In other words, the chapters elaborate on the verification procedure outlined as part of our research methodology in Section 1.5. An evaluation of this verification procedure is presented in this chapter. Here, we examine the tests and analysis conducted on the proposed solution and the inferences that can be drawn from them in order to determine whether or not the solution completely fulfills the requirements for which it has been designed.

5.1 Service Composition Process Evaluation

As stated in Section 1.2, the first objective of this thesis is to design an operational service composition mechanism that can generate one or more constraint- and context-aware composite services as solutions to a valid composition problem, depending on the services available for use as components during composition. We describe the planning-graph-based composition approach designed by Laleh et al. [4, 5, 6, 7, 8] upon which we base our implementation of this mechanism in Chapter 3 together with the various modifications that we introduce into the original technique in order to complete and optimize it while transforming it into a generic, flexible and maintainable application, as intended. In this section, we explain the evaluation technique employed to ensure that this application fulfills all its functional requirements, i.e., it achieves our first thesis objective.

As explained in Chapter 3, our service composition method is divided into several stages – forward expansion, backward search, solution plan construction, constraint-aware plan construction with optimally adjusted constraints and service composition. Each of these stages has a specific goal; for instance, the forward expansion stage aims at generating a valid search graph whereas the backward search stage is focused on extracting valid solution plan sets from that search graph. In order to achieve its goal effectively, each of these stages needs to fulfill a set of specific conditions and exhibit certain properties during its processing. As part of our evaluation technique, we prepare exhaustive lists of all such defining characteristics and perform tests on each stage individually as well as all stages combined as a process to ensure that all the required conditions are met. While we understand that such a scenario-based testing technique is not an absolute proof of absence of faulty behavior and its effectiveness is contingent upon the thoroughness with which the test cases are designed, the intricacies of our composition process and time restrictions place preparation of a full-fledged mathematical proof, evaluating each constituent operation and possible scenario, outside the scope of this thesis. Nevertheless, we attempt to ensure to the best of our abilities that all essential features of the composition process are thoroughly tested by adopting a meticulous and systematic approach towards the design and execution of the test cases.

The essential properties tested for the forward expansion stage along with the significance of each property, the inputs used for testing it and the expected/actual output obtained that proves that the property has been correctly incorporated in the composition process implementation have been listed in Table 7. Tables 8, 9, 10 and 11 summarize similar tests conducted and results obtained for each of the other stages involved in the process.

Table 7: Forward Expansion Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composition Request (R)	**Service Repository**	**Input Characteristics***	
1	A service from the repository should be added to a search graph only if all its input parameters are available in $prdSet$ (either as initial parameters $R.I$ or as outputs of services present in the already-constructed layers of the graph).	Algorithm 2, lines 4 - 6	**R.I** $= \{int : input11, boolean : input21,\ int : input22, string : input31,\ boolean : input32\}$ **R.O** $= \{char : output32, string : output71\}$	Table 12	$sname2, 3, 7$: all inputs $sname1, 4, 6, 8, 10, 11$: partial $sname5$: no inputs available $sname9$: other failure	Valid search graph composed of services $sname2$, $sname3$ and $sname7$ created
2	A service from the repository should be added to a search graph only if it produces at least one output parameter that does not already exist in $prdSet$ at the time of its addition.	Algorithm 2, lines 4 - 6 Also, see Section 3.4	**R.I** $= \{int : input11,\ char : input12, boolean : input21,\ int : input22, string : input31,\ boolean : input32\}$ **R.O** $= \{char : output32, string : output71\}$	Table 12	$sname1, 2, 3, 7$: new output $sname8, 9$: no new output $sname4, 5, 6, 10, 11$: failure	Valid search graph composed of services $sname1$, $sname2$, $sname3$ and $sname7$ created
3	The same service should not be added multiple times to a search graph.	Result of Property #2 Detailed in Section 3.4	**R.I** $= \{int : input11,\ char : input12, boolean : input21,\ int : input22, string : input31,\ boolean : input32\}$ **R.O** $= \{char : output32, string : output71\}$	Table 12	$sname1$ can be added to $Layer$ 0 and $Layer$ 2 (as successor to $sname7$ in $Layer$ 1)	Valid search graph with $sname1$ only in $Layer$ 0 and $sname7$ in $Layer$ 1 created
4	Component services with different names but same input-output specifications should be allowed to be added to the same search graph.	Allows alternative solution plans to be constructed for a given composition request	**R.I** $= \{int : input11, boolean : input21,\ int : input22, string : input42,\ char : input61\}$ **R.O** $= \{int : output42, char : output61\}$	Table 12	$sname6, 10$: same I/O specs $sname2, 4, 6, 10$: valid $other\ services$: some failure	Valid search graph composed of services $sname2$, $sname4$, $sname6$ and $sname10$ created
5	If there are no services in the repository whose inputs are completely available in the set of initial parameters $R.I$, forward expansion should fail.	Shows unavailability of required component services	**R.I** $= \{int : inputXX, char : inputYY\}$ **R.O** $= \{int : output42, string : output71\}$	Table 12	No service in the repository accepts $inputXX$ or $inputYY$ as input	No search graph created. Forward expansion fails.
6	If the component services of a search graph cannot collectively produce all the goal parameters $R.O$, forward expansion should fail.	Shows that the given composition request cannot be served by the available services	**R.I** $= \{int : input11,\ char : input12, boolean : input21,\ int : input22, string : input31,\ boolean : input32\}$ **R.O** $= \{string : outputXX, boolean : outputYY\}$	Table 12	No service in the repository produces $outputXX$ or $outputYY$ as output	No search graph created. Forward expansion fails.
7	If a search graph composed of only one service gets created successfully, forward expansion should fail.	Implies that request can be served by one service only. No composition can be created from available services.	**R.I** $= \{boolean : input21, int : input22\}$ **R.O** $= \{char : output21, float : output22\}$	Table 12	$sname2$ serves the given composition request completely. No composition of available services fulfills the request.	Search graph composed of only $sname2$ rejected. Forward expansion fails.

* Only characteristics relevant to each test case have been specified in their corresponding entry, for simplicity and lack of space. Nevertheless, services that get added to a search graph fulfill all required conditions (whether explicitly listed in the entry or not).

Table 8: Backward Search Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composition Request (R)	Service Repository	Input Characteristics*	
1	Each layer of a search graph should be successively processed as the starting layer for a backward search iteration.	Algorithm 1, lines 3 - 11 Allows all possible plan sets to be generated	**R.I** $= \{int : input11,\ char : input12,\ string : input31,\ string : input42,\ char : input61,\ float : output22\}$ **R.O** $= \{float : output11,\ boolean : input32\}$	Table 12	$SG\ Layer\ 0:\ sname1, 4, 8$ $SG\ Layer\ 1:\ sname3, 6, 7$	Valid plan sets created starting from $Layer\ 0$ $(\{sname1, 8\}, \{1, 4, 8\})$ and from $Layer\ 1$ $(\{sname8, 3, 7\})$
2	A service set from a starting layer in a search graph that does not produce any requested output should immediately be discarded as invalid and not be allowed to proceed any further with the backward search.	Prevents generation of a solution plan whose last layer does not produce any goal parameter and, hence, must be pruned.	**R.I** $= \{char : input12,\ string : input31,\ boolean : input32\}$ **R.O** $= \{float : output11\}$	Table 12	$SG\ Layer\ 0:\ sname3$ $SG\ Layer\ 1:\ sname7$ $SG\ Layer\ 2:\ sname1, 8$ $output11 \notin sname8.O$	Service set $\{sname8\}$ from starting layer 2 discarded. Valid sets $\{sname3, 7\}, \{3, 7, 1\}, \{3, 7, 1, 8\}$ created.
3	Power sets of all services in each layer of a search graph should be created and each element of those power sets should be processed.	Algorithm 1, lines 3 - 11 Algorithm 4, lines 3 - 4 Allows all possible plan sets to be generated	**R.I** $= \{int : input11,\ char : input12,\ boolean : input21,\ int : input22,\ string : input31,\ string : input42,\ char : input61,\ float : output22\}$ **R.O** $= \{float : output11,\ boolean : input32\}$	Table 12	$SG\ Layer\ 0:\ sname1, 2, 8$ $SG\ Layer\ 1:\ sname3, 4$ $SG\ Layer\ 2:\ sname6, 7, 10$	All 12 valid plan sets created, containing 2 to 7 services each and covering all power set combinations
4	A plan set extracted from a search graph should be considered valid only if it produces all the goal parameters ($R.O$). Invalid plan sets should be discarded.	Algorithm 4, lines 16 - 21 Allows early rejection of service sets incapable of serving the given composition request	**R.I** $= \{string : input31,\ boolean : input32,\ string : input42,\ char : output21\}$ **R.O** $= \{float : output22,\ char : output32,\ int : output42\}$	Table 12	$SG\ Layer\ 0:\ sname3$ $SG\ Layer\ 1:\ sname7, 9$ $SG\ Layer\ 2:\ sname4$	$\{sname3, 9, 4\}$ created as valid plan set. $\{sname3, 9\},\ \{3, 7, 9\}$ discarded for not producing $output42$.
5	A plan set comprising of only one service should be discarded as invalid.	Such a plan set does not qualify as a composition of services.	**R.I** $= \{char : input12,\ int : output31\}$ **R.O** $= \{float : output11\}$	Table 12	$SG\ Layer\ 0:\ sname7$ $SG\ Layer\ 1:\ sname1, 8$	$\{sname7, 1\},\ \{7, 1, 8\}$ created as valid plan sets. $\{sname7\}$ discarded for containing only one service.
6	A plan set may be composed of services all of which belong to the same layer, which may eventually result in a solution plan comprising of just one layer containing multiple component services.	Unique yet completely valid case that must be allowed	**R.I** $= \{int : input11,\ char : input12,\ string : input31,\ string : input42,\ char : input61,\ float : output22\}$ **R.O** $= \{float : output11,\ boolean : input32\}$	Table 12	$SG\ Layer\ 0:\ sname1, 4, 8$ $SG\ Layer\ 1:\ sname3, 6, 7$	Valid plan sets $\{sname1, 8\}, \{1, 4, 8\}$ created with services just from $Layer\ 0$
7	Multiple plan sets may be generated from a search graph, including those that are just partially different from each other.	Maximizes the number of alternative solutions generated for a composition request	**R.I** $= \{char : input12,\ string : input31,\ boolean : input32\}$ **R.O** $= \{float : output11\}$	Table 12	$SG\ Layer\ 0:\ sname3$ $SG\ Layer\ 1:\ sname7$ $SG\ Layer\ 2:\ sname1, 8$	Partially different yet completely valid plan sets $\{sname3, 7\},\ \{3, 7, 1\},\ \{3, 7, 1, 8\}$ created

* SG stands for "Search Graph".

Table 9: Plan Construction Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output*
			Composition Request (R)	Service Repository	Input Characteristics	
1	A service whose inputs are not completely available in a given solution plan (possibly due to removal of its predecessors from the plan as invalid nodes) should be removed from the plan.	Algorithm 5, lines 10 - 17 Also, see Section 3.5.3.3	**R.I** $= \{int : input11, string : input21\}$ **R.O** $= \{int : output131\}$	Table 13	15 plan sets created, comprising of various combinations of all 13 services in the repository	$sname9, 10, 11, 12, 13$ removed from 14 plan sets for failing this condition. Valid solution plan created: $[0 : 1, 2; 1 : 9, 10; 2 : 11, 12; 3 : 13]$
2	A service with no successors in a given solution plan should produce at least one goal parameter or else be removed from the plan.	Algorithm 5, lines 23 - 27 Also, see Section 3.5.3.3	**R.I** $= \{int : input11, char : input12, boolean : input21, int : input22, string : input31, string : input42, char : input61, float : output22\}$ **R.O** $= \{float : output11, boolean : input32\}$	Table 12	12 plan sets created, including $\{sname1, 2, 8\}$, $\{2, 8, 3, 4, 6, 7\}$, $\{2, 8, 3, 4, 7, 10\}$, $\{2, 8, 3, 4, 6, 7, 10\}$	$sname2, 4, 6, 10$ removed from these plan sets for failing this condition. Valid solution plans created: $[0 : sname1, 8]$, $[0 : sname8; 1 : 3; 2 : 7]$.
3	A solution plan should be validated repetitively until no more invalid services could be found and removed from it.	Algorithm 5, lines 6 - 30 Ensures complete removal of unnecessary services	Same as Property #2	Table 12	12 plan sets created, comprising of various combinations of $sname1, 2, 3, 4, 6, 7, 8, 10$	10 resultant solution plans pruned in first iteration, found valid in second, later discarded as duplicates.
4	A solution plan that does not produce all the goal parameters ($R.O$) after pruning is complete should be discarded as invalid.	Algorithm 5, lines 32 - 39 Ensures that the given composition request can be served by the plan	**R.I** $= \{int : input11, float : input12, string : input21, boolean : input22\}$ **R.O** $= \{int : output71, float : output81\}$	Table 13	368 valid plan sets created. $sname7, 8$ produce $output71$ and $output81$ respectively.	All resultant plans other than $[0 : sname1; 1 : 3, 4; 2 : 7, 8]$, $[0 : 1, 2; 1 : 3, 4; 2 : 7, 8]$ and $[0 : 2; 1 : 3, 4; 2 : 7, 8]$ discarded.
5	A solution plan comprising of only one service should be discarded as invalid.	Such a solution plan does not qualify as a composition of services	**R.I** $= \{int : input11, float : input12, string : input21, boolean : input22\}$ **R.O** $= \{char : output41\}$	Table 13	20 plan sets created. $sname4$ takes $input21, 22$ and produces $output41$.	6 plan sets result in solution plan $[0 : \emptyset; 1 : sname4]$, discarded for containing only 1 service.
6	Duplicate solution plans constructed for a composition request should be discarded.	Algorithm 1, lines 7 - 10 Prevents redundant processing of same plan	Same as Property #2	Table 12	12 plan sets created, comprising of various combinations of $sname1, 2, 3, 4, 6, 7, 8, 10$	Valid plans $[0 : sname1, 8]$, $[0 : sname8; 1 : 3; 2 : 7]$ created. Other 10 resultant plans discarded as duplicates after pruning.
7	A solution plan may be composed of two or more parallel branches that share no services among themselves.	Unique yet completely valid case that must be allowed	**R.I** $= \{int : input11, float : input12, string : input21, boolean : input22\}$ **R.O** $= \{string : output51, boolean : output52\}$	Table 13	884 valid plan sets created. $sname5, 6$ produce $output51$ and $output52$.	Out of the 7 resultant plans, 3 have parallel branches composed of $\{sname1, 3, 5\}$ & $\{2, 4, 6\}$, $\{1, 3, 5\}$ & $\{4, 6\}$ and $\{3, 5\}$ & $\{2, 4, 6\}$.
8	A solution plan, at this stage, may contain an empty service layer as a result of pruning. Such layers should be removed in the next stage.	Unique yet completely possible scenario that must be handled	**R.I** $= \{int : input11, float : input12, string : input21, boolean : input22\}$ **R.O** $= \{string : output51, boolean : output52\}$	Table 13	884 valid plan sets created. $sname5, 6$ produce $output51$ and $output52$.	Out of the 7 resultant solution plans, $Layer$ 0 is empty for $[0 : \emptyset; 1 : sname3; 2 : 5]$ and $[0 : \emptyset; 1 : 4; 2 : 6]$.

* Solution plans are represented as [list of *layer index* : *services in that layer*].
Due to lack of space, service names are shortened to their indexes. For instance, $[0 : 1, 2]$ in test case 1 represents $[Layer\ 0 : sname1,\ sname2]$.

Table 10: Constraint-aware Plan Construction Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output*
			Composition Request (R)	Service Repository	Input Characteristics*	
1	Predecessor and successor lists of a service-node in a constraint-aware plan should have pointers to only those nodes that exist in the plan. Other irrelevant predecessors and successors should be removed from their lists for that plan.	Optimization operation Detailed in Section 3.5.3.4	**R.I** $= \{int : input11, float : input12, string : input21, boolean : input22\}$ **R.O** $= \{string : output51, boolean : output52\}$	Table 13	2 of the 7 plans created: P1: $[0 : 2; 1 : 4; 2 : 6]$, P2: $[0 : 1; 1 : 3, 4; 2 : 5, 6]$	*sname*2 removed from *sname*4's predecessor list for plan P2 but not P1. Also, *sname*8 (successor to *sname*4 in the search graph) removed from *sname*4's successor lists.
2	Empty service layers (irrespective of their count or location) should be removed from constraint-aware plans.	Optimization operation Detailed in Section 3.5.3.4	Not used	Not used	Manually generated plan: $[2 : 1; 4 : 2, 3; 7 : 4, 5, 6; 8 : 7; 10 : 8, 9]$. *Layers* 0, 1, 3, 5, 6, 9 are empty.	All 6 empty layers removed and remaining layer indexes rearranged. Resultant plan: $[0 : 1; 1 : 2, 3; 2 : 4, 5, 6; 3 : 7; 4 : 8, 9]$.
3	Constraint adjustment should be performed for every constraint of every service-node (other than those in the first layer) in a constraint-aware plan.	Algorithm 6, lines 1 - 9 Ensures optimum internal constraint placement	**R.I** $= \{int : input111\}$ **R.O** $= \{char : output151\}$	Table 14	Solution plan created: $[0 : 11; 1 : 12, 13; 2 : 14; 3 : 15]$	Each constraint on each service-node in the plan, which is not optimally located, is adjusted.
4	Constraint adjustment should not be performed for the constraints placed on service-nodes belonging to the first layer of a constraint-aware plan.	Algorithm 6, lines 1 - 2 No predecessors to these nodes, which could affect these constraint-features	**R.I** $= \{int : input111, boolean : input171\}$ **R.O** $= \{boolean : output241, string : output231\}$	Table 14	Plan created: $[0 : 11, 17; 1 : 12, 20; 2 : 19, 21, 22; 3 : 23, 24]$	Constraints placed on $W11$ (on $i111$) and $W17$ (on $i111, i171$) not affected during adjustment.
5	An adjusted constraint should be added to all the successors of the service-node that last affects the constraint feature in the plan, irrespective of whether the successor uses the feature or not.	Algorithm 6, lines 15 - 19 Minimizes rollback effort in case of run-time constraint violation	Same as Property #3	Table 14	Same plan as #3. $W11$ produces $o111, o112$, $W12$ accepts $o111$, $W13$ accepts $o112$. $W12, 13$ are successors to $W11$.	After adjustment, $W12$'s constraint on $o111$ is attached to $W13$ as well. Similar adjustment done for constraints on other service-nodes as well.
6	If a service-node whose constraint needs to be adjusted has multiple predecessors, the constraint should be attached to the successors of only that predecessor that affects the constraint feature.	Algorithm 6, lines 15 - 19	**R.I** $= \{int : input111, boolean : input171\}$ **R.O** $= \{string : output231, int : output251, float : output261\}$	Table 14	Plan created: $[0 : 17; 1 : 20; 2 : 21, 22; 3 : 23, 25, 26]$. $W21$ produces $o211$. $W22$ produces $o221$. $W23$ accepts $o211, o221$. $W21$ is predecessor to $W23, 26$ and $W22$ is to $W23, 25$.	After adjustment, $W23$'s constraint on $o221$ is attached to $W25$ as well but not to $W26$.
7	A constraint on a feature that is not affected by any service in a constraint-aware plan should be moved to the beginning of the plan.	Algorithm 6, lines 10 - 26	**R.I** $= \{string : DeliveryAddress, string : ProductName\}$ **R.O** $= \{string : ShipmentConfirm\}$	Table 14	15 valid plans created. Each contains $W3, 4, 7$, which have constraints on *DeliveryAddress* – not affected by any service.	After adjustment, constraints on *DeliveryAddress* are moved and attached only to $W1$ – the only service in *Layer* 0 of the plans.

8	A constraint moved to the beginning of a plan during adjustment should be attached to all the service-nodes in the first layer of the plan.	Algorithm 6, lines 10 - 26 Minimizes rollback effort in case of run-time constraint violation	**R.I** = $\{int : input111,$ $boolean : input171\}$ **R.O** = $\{boolean : output241,$ $string : output231\}$	Table 14	Plan created: $[0 : 11, 17;$ $1 : 12, 20; 2 : 19, 21, 22;$ $3 : 23, 24]$. *input*181 not affected by any service.	After adjustment, $W19$'s constraint on $i181$ is moved and attached to both $W11$ and $W17$ in *Layer* 0 of the plan.
9	The number of layers between a predecessor and its successor node should not affect the constraint adjustment process.	Ensure optimum internal constraint placement	**R.I** = $\{string : DeliveryAddress,$ $string : ProductName\}$ **R.O** = $\{string : ShipmentConfirm\}$	Table 14	15 valid plans created. e.g.: $[0 : 1; 1 : 2; 2 : 3, 4]$. *ProductAddress* is last produced/affected by $W1$ (predecessor to $W2, 3, 4$).	After adjustment, $W3$'s constraint on *ProductAddress* is attached to $W2$ and $W4$ as well irrespective of the layers to which they belong.
10	Multiple copies of a constraint (i.e., *Constraint* JAVA object) should not be attached to a service-node in a plan.	Optimization operation Detailed in Section 3.5.3.4	Same as Property #9	Table 14	Same as Property #9. Also, $W3$ has a constraint (represented as $C31$) on *ProductAddress*.	Before adjustment, $W3$ has 1 $C31$ object. During adjustment, another $C31$ is meant to be attached to $W3$ (successor of $W1$), but it is not, being a duplicate.
11	In case all service constraints in a plan are already optimally located, no adjustment would be required for them.	Unique yet completely valid case that must be allowed	**R.I** = $\{int : input111\}$ **R.O** = $\{string : output161\}$	Table 14	Plan created: $[0 : 11;$ $1 : 12; 2 : 16]$. $i111$: initial parameter, accepted by $W11$. $o111$: produced by $W11$, accepted by $W12$. $o121$ produced by $W12$, accepted by $W16$.	All constraints in the plan – $W11$'s constraint on $i111$, $W12$'s constraints on $o111$, $W16$'s constraints on $o121$ – are optimally located to begin with. No constraint adjustment made.
12	In case no service-node in a constraint-aware plan has any constraint imposed on it, the constraint segment of each service-node in the plan should be blank/empty.	Unique yet completely valid case that must be allowed	**R.I** = $\{string : StudentID\}$ **R.O** = $\{float : MarksPercentage\}$	Table 14	Solution plan created: $[0 : W8, 1 : W9; 2 : W10]$. $W8, 9, 10$ do not have any constraints.	Optimized constraint-aware plan created with empty constraint segments for all its service-nodes

* Solution plans are represented as [list of *layer index* : *services in that layer*].
Due to lack of space, service names are abbreviated to their indexes. For instance, $[0 : 2]$ in test case 1 represents $[Layer\ 0 : sname2]$ while $[0 : 11]$ in test case 3 represents $[Layer\ 0 : W11]$.
Similarly, parameter names have also been shortened from *input* and *output* to i and o respectively. For instance, $o111, o112$ in test case 4 represent $output111, output112$ respectively.

Table 11: Service Composition Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output*
			Composition Request (R)	**Service Repository**	**Input Characteristics***	
1	A composition request that does not specify any input/initial parameters ($R.I$) should be considered invalid.	Planning-graph service composition is based on input-output relationship between services	$\mathbf{R.I} = \emptyset$ $\mathbf{R.O} = \{string : ShipmentConfirm, string : Invoice\}$	Irrelevant	User request specifies no input parameters. Other request elements are irrelevant in this case.	Request discarded as invalid. "No initial parameter" error logged. Composition process aborted.
2	A composition request that does not specify any output/goal parameters ($R.O$) should be considered invalid.	Planning-graph service composition is based on input-output relationship between services	$\mathbf{R.I} = \{string : DeliveryAddress, string : ProductName, string : CustomerName, float : Price\}$ $\mathbf{R.O} = \emptyset$	Irrelevant	User request specifies valid input but no output parameters. Other req-uest elements irrelevant.	Request discarded as invalid. "No goal parameter" error logged. Composition process aborted.
3	A composition request that specifies a QoS feature other than *COST*, *RESPONSE_TIME, RELIABILITY, AVAILABILITY* should be considered invalid.	Ensures compliance with Laleh's composition model [5]	**R.I** = Same as Property #2 **R.O** = Same as Property #1 $\mathbf{R.QoS} = \{COST, THROUGHPUT, response\_time\}$	Irrelevant	User request specifies valid inputs and outputs but invalid QoS features. Other request elements are irrelevant in this case.	Request discarded as invalid. "Unidentified QoS feature" error logged. Composition process aborted.
4	A composition request that specifies a constraint placed on a feature other than the inputs, outputs and QoS features specified in the request should be considered invalid.	Ensures compliance with Laleh's composition model [4]	**R.I** = Same as Property #2 **R.O** = Same as Property #1 $\mathbf{R.QoS} = \{COST\}$ $\mathbf{R.C} = \{AVAILABILITY = 60, string : DeliveryAddress = Quebec, string : Invoice = true\}$	Irrelevant	User request specifies valid inputs, outputs and QoS features but invalid feature in first constraint. Other request elements are irrelevant in this case.	Request discarded as invalid. "Invalid constraint feature" error logged. Composition process aborted.
5	A composition request that specifies a constraint composed of less than or more than 3 elements (feature, operator, literal value) should be considered invalid.	Ensures compliance with Laleh's composition model [4]	**R.I** = Same as Property #2 **R.O** = Same as Property #1 **R.QoS** = Same as Property #4 $\mathbf{R.C} = \{COST <, string : DeliveryAddress = Quebec\ City\}$	Irrelevant	User request specifies valid inputs, outputs and QoS features but 2 eleme-nts for first and 4 elemen-ts for second constraint. Other elements irrelevant.	Request discarded as invalid. "Invalid constraint format" errors logged. Composition process aborted.
6	A composition request that specifies a constraint with an operator other than $<, >, =, <=, >=$ should be considered invalid.	Ensures compliance with Laleh's composition model [4] and service repository framework [51]	**R.I** = Same as Property #2 **R.O** = Same as Property #1 **R.QoS** = Same as Property #4 $\mathbf{R.C} = \{string : Invoice <> false\}$	Irrelevant	User request specifies valid inputs, outputs and QoS features but invalid operator for constraint. Other request elements are irrelevant in this case.	Request discarded as invalid. "Invalid constraint operator" error logged. Composition process aborted.
7	Service composition process should be aborted immediately in case the composition request is found invalid.	Valid composition request is mandatory trigger condition for composition process	Same as any of Properties #1 - #6	Irrelevant	One or the other issue with composition request as listed for Properties #1 - #6.	Request discarded as invalid. Appropriate error messages logged. Composition process aborted immediately.

8	Service composition process should be aborted immediately in case the service repository specified by the user is empty, i.e., no services are available.	Service availability is mandatory requirement for composition process	**R.I** $= \{string : DeliveryAddress,$ $string : ProductName\}$ **R.O** $= \{string : ShipmentConfirm\}$ **R.QoS** $= \{COST\}$ **R.C** $= \{COST < 100,$ $string : DeliveryAddress = Canada\}$	Empty XML service repository	User composition request is valid but user-specified service repository is empty.	"Empty service repository" error logged. Composition process aborted immediately.
9	Service composition process should fail in case the given composition request cannot be served by the services in the given service repository.	Ensures that composition process as a whole functions as expected in case of failure	**R.I** $= \{string : DeliveryAddress,$ $string : ProductName\}$ **R.O** $= \{string : Invoice\}$	Table 14	None of the services in the given repository produce $Invoice$ as an output parameter.	"Unsolvable composition problem" error logged. No constraint-aware plans generated.
10	Service composition process should fail in case the given composition request can be served by a single service from the given repository and no composition can be constructed to serve it.	Ensures that composition process as a whole functions as expected in case of failure	**R.I** $= \{string : DeliveryAddress,$ $string : ProductName\}$ **R.O** $= \{string : ProductNumber\}$	Table 14	Only 1 valid plan set can be created: $\{W1\}$.	"Solvable by atomic service" error logged. No constraint-aware plans generated.
11	A composite service may contain sequential, parallel, split or join type of component service arrangement. These arrangements may exist individually or in combination with each other within the composite service.	Ensures all acceptable arrangements of component services can be generated	**R.I** $= \{int : input111,$ $boolean : input171\}$ **R.O** $= \{boolean : output241,$ $string : output231\}$	Table 14	Constraint-aware plan created: $[0 : 11, 17;$ $1 : 12, 20; 2 : 19, 21, 22;$ $3 : 23, 24]$	Plan has sequential arrangement of $W11, 12, 19, 24$; parallel of $11, 12, 19, 24$ and $17, 20, 21, 22, 23$; split from 20 to $21, 22$; join from $21, 22$ to 23.
12	If required for storage, a constraint-aware plan should first be correctly transformed into a layered composite service object.	Allows storage and re-use of generated compositions	**R.I** $= \{string : StudentID\}$ **R.O** $= \{float : MarksPercentage\}$	Table 14	Constraint-aware plan created: $\{0 : W8; 1 : 9;$ $2 : 10\}$.	Layered composite service name starts with "CompSvc_"; its inputs and outputs are same as those of the given request; its effects and constraints are sets of those of $W8, 9, 10$; its plan is same as the source plan.
13	If so requested by the user, all the constraint-aware plans generated for a composition request should be stored in the given service repository as layered composite service objects and be available for use as components for any future compositions.	Allows storage and re-use of generated compositions	**Request #1:** **R.I** $= \{string : StudentID\}$ **R.O** $= \{float : MarksPercentage\}$ **CS Storage Flag** $= Y$ **Request #2:** **R.I** $= \{string : StudentID\}$ **R.O** $= \{char : Grade\}$	Table 15 in serialized Java object format	Given XML repository is first translated into a serialized format, which is then used as input for Request #1. This repository appended with $CS8910$ is used as input for Request #2.	Plan created for Request #1: $\{0 : W8; 1 : 9; 2 : 10\}$, which is stored as composite service (represented as $CS8910$) in given serialized repository. Plan created for Request #2: $\{0 : CS8910, 1 : 11\}$ instead of $\{0 : 8; 1 : 9; 2 : 10; 3 : 11\}$.

* Solution plans are represented as [list of *layer index* : *services in that layer*].
Due to lack of space, service names are abbreviated to their indexes. For instance, $[0 : 11, 17]$ in test case 11 represents $[Layer\ 0 : W11,\ W17]$.

Table 12: Services Available for Composition in Test Repository 1

#	Service Name	Service Inputs	Service Outputs	Internal Service Constraints	Service Effects
1	$sname1$	$int : input11$, $char : input12$	$float : output11$, $string : output12$	$int : input11 < 11$, $char : input12 = a$	$float : output11$, $string : output12$
2	$sname2$	$boolean : input21$, $int : input22$	$char : output21$, $float : output22$	$boolean : input21 = true$, $int : input22 < 22$	$char : output21$, $float : output22$
3	$sname3$	$string : input31$, $boolean : input32$	$int : output31$, $char : output32$	$string : input31 = lit31$, $boolean : input32 = false$	$int : output31$, $char : output32$
4	$sname4$	$float : output22$, $string : input42$	$boolean : output41$, $int : output42$	$float : output22 > 41.0$, $string : input42 = lit42$	$boolean : output41$, $int : output42$
5	$sname5$	$char : input51$, $float : input52$	$string : output51$, $boolean : output52$	$char : input51 = b$, $float : input52 < 5.2$	$string : output51$, $boolean : output52$
6	$sname6$	$boolean : output41$, $char : input61$, $float : output22$, $int : input11$	$string : output51$, $int : output42$, $char : output61$	$boolean : output41 = true$, $char : input61 = c$	$string : output51$, $int : output42$, $char : output61$
7	$sname7$	$int : output31$	$string : output71$, $float : output11$, $int : input11$	$int : output31 > 71$, $int : output31 < 72$	$string : output71$, $float : output11$, $int : input11$
8	$sname8$	$int : input11$, $char : input12$	$boolean : input21$, $boolean : input32$	$int : input11 = 81$, $char : input12 = f$	$boolean : input21$, $boolean : input32$
9	$sname9$	$char : output21$, $int : output31$	$float : output22$	$char : output21 = g$, $int : output31 <= 92$	$float : output22$
10	$sname10$	$boolean : output41$, $char : input61$, $float : output22$, $int : input11$	$string : output51$, $int : output42$, $char : output61$	$boolean : output41 = false$, $char : input61 = h$	$string : output51$, $int : output42$, $char : output61$
11	$sname11$	$char : output61$, $int : input11$	$int : output42$, $char : output61$	$char : output61 = i$, $int : input11 = 112$	$int : output42$, $char : output61$

Table 13: Services Available for Composition in Test Repository 2

#	**Service Name**	**Service Inputs**	**Service Outputs**	**Internal Service Constraints**	**Service Effects**
1	$sname1$	$int : input11$	$float : input12$, $char : output11$	$int : input11 <= 11$, $int : input11 >= 5$	$float : input12$, $char : output11$
2	$sname2$	$string : input21$	$boolean : input22$, $int : output21$	$string : input21 = lit21$	$boolean : input22$, $int : output21$
3	$sname3$	$int : input11$, $float : input12$	$float : output31$	$int : input11 = 31$, $float : input12 < 32.2$	$float : output31$
4	$sname4$	$string : input21$, $boolean : input22$	$char : output41$	$string : input21 = lit41$, $boolean : input22 = false$	$char : output41$
5	$sname5$	$float : output31$	$string : output51$, $boolean : output52$	$float : output31 > 51.1$, $float : output31 < 52.2$	$string : output51$, $boolean : output52$
6	$sname6$	$char : output41$	$string : output51$, $boolean : output52$	$char : output41 = x$	$string : output51$, $boolean : output52$
7	$sname7$	$float : output31$	$int : output71$	$float : output31 >= 71.0$, $float : output31 < 72.0$	$int : output71$
8	$sname8$	$char : output41$	$float : output81$	$char : output41 = y$	$float : output81$
9	$sname9$	$char : output11$, $int : output21$	$char : output91$	$char : output11 = z$, $int : output21 < 92$	$char : output91$
10	$sname10$	$int : output21$	$string : output101$	$int : output21 > 101$, $int : output21 < 102$	$string : output101$
11	$sname11$	$char : output91$, $string : output101$	$char : output111$	$char : output91 = w$, $string : output101 = lit112$	$char : output111$
12	$sname12$	$string : output101$	$boolean : output121$	$string : output101 = lit121$	$boolean : output121$
13	$sname13$	$char : output111$, $boolean : output121$	$int : output131$	$char : output111 = u$, $boolean : output121 = true$	$int : output131$

Table 14: Services Available for Composition in Test Repository 3

#	Service Name	Service Inputs	Service Outputs	Internal Service Constraints	Service Effects
1	$W1$	$string : ProductName$, $string : DeliveryAddress$	$int : ProductNumber$, $string : ProductAddress$	$\emptyset$	$int : ProductNumber$, $string : ProductAddress$
2	$W2$	$int : ProductNumber$	$int : PaymentNumber$, $int : OrderNumber$	$\emptyset$	$int : PaymentNumber$, $int : OrderNumber$
3	$W3$	$int : PaymentNumber$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Montreal$, $string : ProductAddress = Montreal$	$string : ShipmentConfirm$
4	$W4$	$int : PaymentNumber$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Quebec$, $string : ProductAddress = Quebec$	$string : ShipmentConfirm$
5	$W5$	$int : ProductNumber$	$int : OrderNumber$	$\emptyset$	$int : OrderNumber$
6	$W6$	$int : ProductNumber$	$string : PaymentConfirm$	$\emptyset$	$string : PaymentConfirm$
7	$W7$	$string : PaymentConfirm$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Canada$, $string : ProductAddress = Canada$	$string : ShipmentConfirm$
8	$W8$	$string : StudentID$	$float : TotalMarks$, $int : NumberOfCourses$	$\emptyset$	$float : TotalMarks$, $int : NumberOfCourses$
9	$W9$	$float : TotalMarks$, $int : NumberOfCourses$	$float : AverageMarks$	$\emptyset$	$float : AverageMarks$
10	$W10$	$float : AverageMarks$	$float : MarksPercentage$	$\emptyset$	$float : MarksPercentage$
11	$W11$	$int : input111$	$float : output111$, $char : output112$	$int : input111 <= 111$	$float : output111$, $char : output112$
12	$W12$	$float : output111$	$string : output121$	$float : output111 > 121.0$, $float : output111 < 122.0$	$string : output121$
13	$W13$	$char : output112$	$boolean : output131$, $int : output132$	$\emptyset$	$boolean : output131$, $int : output132$
14	$W14$	$string : output121$, $boolean : output131$	$float : output141$	$string : output121 = lit141$, $boolean : output131 = true$	$float : output141$

15	$W15$	$float : output141$, $int : output132$	$char : output151$	$float : output141 = 15.1$, $int : output132 < 152$	$char : output151$
16	$W16$	$string : output121$	$string : output161$	$string : output121 = lit161$, $string : output121 = lit162$	$string : output161$
17	$W17$	$int : input111$, $boolean : input171$	$int : output171$	$int : input111 = 171$, $boolean : input171 = false$	$int : output171$
18	$W18$	$int : input111$, $float : input181$	$char : output181$	$int : input111 = 181$, $float : input181 < 18.2$	$char : output181$
19	$W19$	$string : output121$	$string : output191$	$string : output121 = lit191$	$string : output191$
20	$W20$	$int : output171$	$boolean : output201$, $int : output202$	$\emptyset$	$boolean : output201$, $int : output202$
21	$W21$	$boolean : output201$	$float : output211$	$\emptyset$	$float : output211$
22	$W22$	$int : output202$	$char : output221$	$\emptyset$	$char : output221$
23	$W23$	$char : output221$, $float : output211$	$string : output231$	$char : output221 = l$	$string : output231$
24	$W24$	$string : output191$	$boolean : output241$	$\emptyset$	$boolean : output241$
25	$W25$	$char : output221$	$int : output251$	$\emptyset$	$int : output251$
26	$W26$	$float : output211$	$float : output261$	$\emptyset$	$float : output261$
27	$W27$	$char : input271$	$string : output271$, $boolean : output272$	$\emptyset$	$string : output271$, $boolean : output272$
28	$W28$	$string : output271$	$int : output281$, $boolean : output272$	$\emptyset$	$int : output281$, $boolean : output272$
29	$W29$	$int : output281$	$float : output291$, $boolean : output272$	$\emptyset$	$float : output291$, $boolean : output272$
30	$W30$	$float : output291$	$char : output301$	$\emptyset$	$char : output301$
31	$W31$	$char : output301$	$string : output311$	$\emptyset$	$string : output311$

Table 15: Services Available for Composition in Test Repository 4

#	Service Name	Service Inputs	Service Outputs	Internal Service Constraints	Service Effects
1	$W1$	$string : ProductName$, $string : DeliveryAddress$	$int : ProductNumber$, $string : ProductAddress$	$\emptyset$	$int : ProductNumber$, $string : ProductAddress$
2	$W2$	$int : ProductNumber$	$int : PaymentNumber$, $int : OrderNumber$	$\emptyset$	$int : PaymentNumber$, $int : OrderNumber$
3	$W3$	$int : PaymentNumber$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Montreal$, $string : ProductAddress = Montreal$	$string : ShipmentConfirm$
4	$W4$	$int : PaymentNumber$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Quebec$, $string : ProductAddress = Quebec$	$string : ShipmentConfirm$
5	$W5$	$int : ProductNumber$	$int : OrderNumber$	$\emptyset$	$int : OrderNumber$
6	$W6$	$int : ProductNumber$	$string : PaymentConfirm$	$\emptyset$	$string : PaymentConfirm$
7	$W7$	$string : PaymentConfirm$, $string : DeliveryAddress$, $string : ProductAddress$, $int : OrderNumber$	$string : ShipmentConfirm$	$string : DeliveryAddress = Canada$, $string : ProductAddress = Canada$	$string : ShipmentConfirm$
8	$W8$	$string : StudentID$	$float : TotalMarks$, $int : NumberOfCourses$	$\emptyset$	$float : TotalMarks$, $int : NumberOfCourses$
9	$W9$	$float : TotalMarks$, $int : NumberOfCourses$	$float : AverageMarks$	$\emptyset$	$float : AverageMarks$
10	$W10$	$float : AverageMarks$	$float : MarksPercentage$	$\emptyset$	$float : MarksPercentage$
11	$W11$	$float : MarksPercentage$	$char : Grade$	$\emptyset$	$char : Grade$
12	$W12$	$float : output111$	$string : output121$	$float : output111 > 121.0$, $float : output111 < 122.0$	$string : output121$
13	$W13$	$char : output112$	$boolean : output131$, $int : output132$	$\emptyset$	$boolean : output131$, $int : output132$
14	$W14$	$string : output121$, $boolean : output131$	$float : output141$	$string : output121 = lit141$, $boolean : output131 = true$	$float : output141$

15	$W15$	$float : output141$, $int : output132$	$char : output151$	$float : output141 = 15.1$, $int : output132 < 152$	$char : output151$
16	$W16$	$string : output121$	$string : output161$	$string : output121 = lit161$, $string : output121 = lit162$	$string : output161$
17	$W17$	$int : input111$, $boolean : input171$	$int : output171$	$int : input111 = 171$, $boolean : input171 = false$	$int : output171$
18	$W18$	$int : input111$, $float : input181$	$char : output181$	$int : input111 = 181$, $float : input181 < 18.2$	$char : output181$
19	$W19$	$string : output121$	$string : output191$	$string : output121 = lit191$	$string : output191$
20	$W20$	$int : output171$	$boolean : output201$, $int : output202$	$\emptyset$	$boolean : output201$, $int : output202$
21	$W21$	$boolean : output201$	$float : output211$	$\emptyset$	$float : output211$
22	$W22$	$int : output202$	$char : output221$	$\emptyset$	$char : output221$
23	$W23$	$char : output221$, $float : output211$	$string : output231$	$char : output221 = l$	$string : output231$
24	$W24$	$string : output191$	$boolean : output241$	$\emptyset$	$boolean : output241$
25	$W25$	$char : output221$	$int : output251$	$\emptyset$	$int : output251$
26	$W26$	$float : output211$	$float : output261$	$\emptyset$	$float : output261$
27	$W27$	$char : input271$	$string : output271$, $boolean : output272$	$\emptyset$	$string : output271$, $boolean : output272$
28	$W28$	$string : output271$	$int : output281$, $boolean : output272$	$\emptyset$	$int : output281$, $boolean : output272$
29	$W29$	$int : output281$	$float : output291$, $boolean : output272$	$\emptyset$	$float : output291$, $boolean : output272$
30	$W30$	$float : output291$	$char : output301$	$\emptyset$	$char : output301$
31	$W31$	$char : output301$	$string : output311$	$\emptyset$	$string : output311$

The tests listed in tables 7 - 11 for the evaluation of our service composition solution have been designed based on a detailed analysis of the planning-graph-based composition technique devised by Laleh et al. [4, 5, 6, 7, 8]. The process of completing, optimizing and transforming their original composition methodology into a flexible and maintainable software application (as described in Chapter 3) has helped us gain valuable insight into its behavioral intricacies and, consequently, a comprehensive knowledge of the essential properties to be exhibited, the exceptional cases to be allowed as well as the pitfalls to be avoided by the composition solution, each of which has been tested thoroughly as part of our evaluation process. While we understand that such a scenario-based testing technique does not prove the solution to be absolutely devoid of faulty behavior, the meticulous study that we have conducted on the composition methodology, the systematic approach that we have adopted towards designing the test cases and the successful execution of all the tests so performed allow us to conclude that, considering our scope and time restrictions, we have been able to effectively evaluate our implemented solution to the best of our capabilities.

5.2 Service Translation Process Evaluation

The primary goal of this thesis is to provide a solution (as described in Contribution **3**) for the verification and validation of constraint- and context-aware composite web services. As discussed in Section 1.2, for this goal to be achieved effectively and efficiently, our proposed solution must fulfill certain specific requirements, which we identify as the objectives of this thesis. In order to fulfill these objectives and, ultimately, accomplish the primary goal of this research, we follow a systematic procedure comprising of a series of clearly-defined tasks to be performed as depicted in Figure 3 and explained in Section 1.5. The first two steps of this procedure (as listed in Section 1.5) are concerned with the construction of service compositions, which forms the first objective of this thesis and is evaluated in Section 5.1. The remaining two steps of the verification procedure – service translation and program execution – need to be performed in order to fulfill the other three objectives of this thesis. An elaborate discussion on the design and implementation of our proposed solution for translating layered composite web services (described in Section 3.5.4.6) into Objective Lucid programs besides other formats is presented in Chapter 4. In this section, we discuss the evaluation technique employed for ensuring that this proposed translation mechanism performs all its functions in accordance with its design goals and specifications. Evaluation of the extent to which the final task in the verification procedure – execution of Objective Lucid representations of service compositions on GIPSY – has been accomplished as part of this research is addressed in brief in Section 5.3.

For evaluating the process of translating layered composite services into Objective Lucid programs, we follow the same strategy as the one adopted for evaluating the service composition process (discussed in Section 5.1). Similar to the composition methodology, we describe our translation process as a set of algorithms in Chapter 4, each of which has a specific goal to be achieved by performing a clearly-defined series of tasks. As part of our evaluation technique, we prepare exhaustive lists of all such constituent operations together with the other required conditions to be met by each of these algorithms and perform tests on each of them individually as well as all of them combined as a process to ensure that the translation solution exhibits all the desired properties. Most of the properties examined as part of this evaluation aim at defining the specific Objective Lucid program construct into which a particular element of a constraint-aware composition plan transforms during the translation process such that when combined and incorporated into a unified procedure, these properties help build a complete and correct Objective Lucid representation of a given layered composite service.

The essential properties tested for Algorithm 8 along with the significance of each property, the inputs used for testing it and the expected/actual output obtained that proves that the property has been correctly incorporated in the translation process implementation have been listed in Table 16. Tables 17, 18, 19 and 20 summarize similar tests conducted and results obtained for Algorithms 9, 10, 11 and 7 respectively.

Table 16: JAVA Segment Generation Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composite Service	Composite Service Inputs	Input Characteristics	
1	JAVA segment of a composite service's OBJECTIVE LUCID translation should begin with a *#JAVA* tag and be composed of JAVA class and free function definitions representing its output accumulator and component service nodes.	Algorithm 8, line 10 Ensures all constituent nodes get represented in JAVA. Also, see Sections 2.1.4 and 4.2.	Figure 22	Not required	Composite service comprising of 2 component services: $W1$ and $W2$	Valid JAVA segment composed of 1 output accumulator class and 2 sets of component class and free function definitions
2	All composite service outputs should be represented as data members of the output accumulator JAVA class irrespective of their count, data type and source component service.	Algorithm 8, line 1 Allows a composite service to have multiple outputs. Explained in Section 2.1.4.	Figure 22	Not required	Composite service with 1 output of *char* data type	Valid *CAWSReqComp* class with 1 *char* type data member
			Figure 24	Not required	Composite service with multiple (3) outputs of *string*, *int* and *boolean* data types generated by 1 component service	Valid *CAWSReqComp* class with 3 data members of types *String*, *int* and *boolean*
			Figure 26	Not required	Composite service with multiple (4) outputs of *string*, *int* and *float* data types generated by multiple component services from different service layers	Valid *CAWSReqComp* class with 2 data members of type *String* and 2 of types *double* and *int*
3	Parameter list of the output accumulator's JAVA class constructor should comprise of all composite service output parameters irrespective of their count, data type and source component service.	Algorithm 8, line 2 Allows a composite service to have multiple outputs. Explained in Section 2.1.4.	Figure 22	Not required	Composite service with 1 output of *char* data type	Valid *CAWSReqComp* constructor with 1 *char* type parameter
			Figure 24	Not required	Composite service with multiple (3) outputs of *string*, *int* and *boolean* data types generated by 1 component service	Valid *CAWSReqComp* constructor with 3 parameters of types *String*, *int* and *boolean*
			Figure 26	Not required	Composite service with multiple (4) outputs of *string*, *int* and *float* data types generated by multiple component services from different service layers	Valid *CAWSReqComp* constructor with 2 parameters of type *String* and 2 of types *double* and *int*
4	Output accumulator's data members should be initialized with the values of their respective parameters accepted by the class constructor irrespective of their count, data type and source component service.	Algorithm 8, line 3 Allows a composite service to have multiple outputs. Explained in Section 2.1.4.	Same as Properties #2 and #3	Not required	Same as Properties #2 and #3	Valid *CAWSReqComp* constructor that initializes all its class data members with their respective parameters

Table 17: Component Service Java Definition Generation Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composite Service	Composite Service Inputs	Input Characteristics	
1	All component service inputs and outputs should be represented as data members of the component's Java class irrespective of their count and data type.	Algorithm 9, lines 1 - 3 Supplies required inputs for service processing in Java. Allows service to have multiple outputs. Explained in Sections 2.1.4 and 4.2.	Figure 22	Not required	$W1$ has 1 *char* type input and 1 *char* type output	Valid *CAWSW1* class with 2 *char* type data members
			Figure 24	Not required	$W3$ has multiple (2) inputs of types *char* and *boolean* and multiple (3) outputs of types *string*, *int* and *boolean*	Valid *CAWSW3* class with 5 data members of types *char*, *boolean*, *String* and *int*
			Figure 26	Not required	$W7$ has 1 *int* type input and 1 *float* type output	Valid *CAWSW7* class with 2 data members of types *int* and *double*
			Figure 23	Not required	$W3$ has multiple (4) inputs and outputs of types *string* and *float*	Valid *CAWSW3* class with 4 data members of types *String* and *double*
2	Parameter list of a component service's Java class constructor should comprise of all of the component's input parameters irrespective of their count, data type and source – user or other component service(s).	Algorithm 9, line 4 Supplies required inputs for service processing in Java. Explained in Sections 2.1.4 and 4.2.	Figure 22	Not required	$W1$ has 1 *char* type input from the user	Valid *CAWSW1* constructor with 1 *char* type parameter
			Figure 26	Not required	$W7$ has 1 *int* type input from 1 component service	Valid *CAWSW7* constructor with 1 *int* type parameter
			Figure 25	Not required	$W1$ has multiple (2) user inputs of types *boolean* and *string*	Valid *CAWSW1* constructor with 2 parameters of types *boolean* and *String*
			Figure 23	Not required	$W3$ has multiple (2) inputs of types *string* and *float* from multiple component services of different service layers	Valid *CAWSW3* constructor with 2 parameters of types *String* and *double*
3	Component service's input data members should be initialized with the values of their respective parameters accepted by the class constructor irrespective of their count, data type and source – user or other component service(s).	Algorithm 9, line 5 Supplies required inputs for service processing in Java. Explained in Sections 2.1.4 and 4.2.	Same as Properties #1 and #2	Not required	Same as Properties #1 and #2	Valid component service constructor that initializes all the input data members of its class with their respective parameters
4	Component service's output data members should be initialized with default values based on their data type by the class constructor irrespective of their count and data type.	Algorithm 9, line 6 Aids clarity of the generated program	Same as Property #1	Not required	Same as Property #1	Valid component service constructor that initializes all the output data members of its class based on their data type – *int* : 0, *double* : 0.0, *char* : ' ', *String* : " " and *boolean* : *false*

5	Component service's output data members should be assigned dummy post-processing values based on their data type by its *process* member method irrespective of their count and data type.	Algorithm 9, line 9 Placeholder processing. Explained in Section 4.2.	Same as Property #1	Not required	Same as Property #1	Valid component service *process* method that assigns dummy values to all the output data members of its class based on their data type – *double* : 20.0, *char* : $'a'$, *String* : “*test*”, *int* : 10 and *boolean* : *true*
6	Parameter list of a component service's JAVA free function should comprise of all of the component's input parameters irrespective of their count, data type and source – user or other component service(s).	Algorithm 9, line 13 Supplies required inputs for service processing in JAVA. Explained in Sections 2.1.4 and 4.2.	Same as Property #2	Not required	Same as Property #2	Valid component service free functions with parameters same as those of the constructors generated for Property #2
7	Component service's JAVA free function should call the service class constructor, passing all its parameters as arguments in the call, to create an object of the service, invoke *process* member method using the object and, finally, return the updated object.	Algorithm 9, lines 14 - 15 Enables LUCID segment to trigger JAVA service processing and receive processed results. Explained in Sections 2.1.4 and 4.2.	Same as Property #2	Not required	Same as Property #2	Valid component service free functions that invoke their respective service constructors with correct arguments, invoke *process* method and return processed object.

Table 18: OBJECTIVE LUCID Segment Generation Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composite Service	Composite Service Inputs	Input Characteristics	
1	LUCID segment of a composite service's OBJECTIVE LUCID translation should begin with a *#OBJECTIVELUCID* tag and be composed of a main expression (representing the service outcome), a global dimension list and output accumulator definition, including all component service definitions.	Algorithm 10, line 17 Drives composite service execution. Ensures all constituent nodes get represented in LUCID. Explained in Sections 2.1.4 and 4.2.	Figure 22	$char : input11 \;=\; 'x'$	Composite service comprises of 2 component services: $W1$ and $W2$, has 1 *char* type input: $input11$ and 1 output: $output21$ produced by $W2$	Valid LUCID segment composed of a main expression evaluated in a 1-D input context, a list of 1 global dimension and LUCID definition of output accumulator evaluated in a 1-D output context, including 2 component service LUCID definitions
2	Global dimension list should comprise of all composite service input parameters irrespective of their count and data type.	Algorithm 10, line 1 Specifies inputs of composite service as its contextual dimensions. Explained in Sections 2.1.4 and 4.2.	Figure 22	$char : input11 \;=\; 'x'$	Composite service has 1 input of *char* data type	Valid list of 1 global dimension
			Figure 24	$int : input11 \;=\; 100$ $float : input21 \;=\; 200.2$	Composite service has multiple (2) inputs of *int* and *float* data types	Valid list of 2 global dimensions
			Figure 25	$boolean : input11 \;=\; true$ $string : input12 \;=\; “xyz”$	Composite service has multiple (2) inputs of *boolean* and *string* data types	Valid list of 2 global dimensions
3	Global evaluation context should comprise of all global dimensions paired with their respective values (enclosed in single quotes if *char* and double quotes if *string* typed) provided by the user irrespective of their count and data type.	Algorithm 10, line 2 Supplies user input to composite service. Enables storage and querying of execution results through GIPSY warehouse.	Same as Property #2	Same as Property #2	Same as Property #2	Valid global context specification with appropriate dimensions and their respective values (enclosed in proper quotes, if required) as provided by the user
4	List of local dimensions and *CAWSReqComp* arguments used in output accumulator's LUCID definition should comprise of all composite service output parameters irrespective of their count, data type and source component service.	Algorithm 10, line 4 Allows a composite service to have multiple outputs. Explained in Section 2.1.4.	Figure 22	$char : input11 \;=\; 'x'$	Composite service with 1 output of *char* data type	Valid accumulator LUCID definition with 1 dimension and 1 *CAWSReqComp* argument
			Figure 24	$int : input11 \;=\; 100$ $float : input21 \;=\; 200.2$	Composite service with multiple (3) outputs of *string*, *int* and *boolean* data types generated by 1 component service	Valid accumulator LUCID definition with 3 dimensions and 3 *CAWSReqComp* arguments
			Figure 26	$int : input11 \;=\; 100$ $float : input21 \;=\; 200.2$	Composite service with multiple (4) outputs of *string*, *int* and *float* data types generated by multiple component services from different service layers	Valid accumulator LUCID definition with 4 dimensions and 4 *CAWSReqComp* arguments

5	Output accumulator's evaluation context should comprise of all its local dimensions paired with their corresponding output data members belonging to the JAVA class of their source component services irrespective of their count, data type and source component service.	Algorithm 10, lines 5 - 10 Allows a composite service to have multiple outputs. Enables storage and querying of execution results through GIPSY warehouse. Explained in Section 2.1.4.	Figure 22	$char : input11 \;=\; 'x'$	Composite service with 1 output of *char* data type	Valid context specification with dimension $l\_output21$ paired with $W2$'s output data member
			Figure 24	$int : input11 \;=\; 100$ $float : input21 \;=\; 200.2$	Composite service with multiple (3) outputs of *string*, *int* *boolean* data types generated by 1 component service	Valid context specification with dimensions $l\_output31$, $l\_output32$ and $l\_output33$ paired with $W3$'s output data members
			Figure 26	$int : input11 \;=\; 100$ $float : input21 \;=\; 200.2$	Composite service with multiple (4) outputs of *string*, *int* and *float* data types generated by multiple component services from different service layers	Valid context specification with dimensions $l\_output31$, $l\_output51$, $l\_output61$ and $l\_output71$ paired with output data members of $W3$, $W5$, $W6$ and $W7$ respectively
6	Output accumulator's LUCID definition should include a *whenever* clause that always evaluates to *true*.	Algorithm 10, line 11 Placeholder for potential user/external service constraints to be introduced in future extensions	Figure 22	$char : input11 \;=\; 'x'$	Composite service with no internal, external or user constraints	Valid LUCID definition of output accumulator with a *whenever* clause that always evaluates to *true*

Table 19: Component Service Lucid Definition Generation Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composite Service	**Composite Service Inputs**	**Input Characteristics**	
1	List of arguments passed from a component service's Lucid definition while calling its Java free function should comprise of all its input parameters irrespective of their count, data type and source – user or other component service(s).	Algorithm 11, line 16 Supplies required inputs for service processing in Java. Explained in Sections 2.1.4 and 4.2.	Figure 22	$char : input11 = 'x'$	$W1$ has 1 *char* type input from the user	Valid Lucid definition of $W1$ with 1 free function argument
			Figure 26	$int : input11 = 100$ $float : input21 = 200.2$	$W7$ has 1 *int* type input from 1 component service	Valid Lucid definition of $W7$ with 1 free function argument
			Figure 25	$boolean : input11 = true$ $string : input12 = “xyz”$	$W1$ has multiple (2) user inputs of types *boolean* and *string*	Valid Lucid definition of $W1$ with 2 free function arguments
			Figure 23	$float : input11 = 200.2$ $float : input41 = 300.3$ $char : input61 = 'x'$	$W3$ has multiple (2) inputs of types *string* and *float* from multiple component services of different service layers	Valid Lucid definition of $W3$ with 2 free function arguments
2	List of local dimensions specified in a component service's Lucid definition should comprise of all its input parameters and features of all the constraints attached to its service-node irrespective of their count, data type, source – user or other component service(s) and of whether their corresponding constraint was added to the service-node during constraint adjustment.	Algorithm 11, lines 1 - 4 Defines a component service's dimensionality. Explained in Section 4.2.	Figure 23	$float : input11 = 200.2$ $float : input41 = 300.3$ $char : input61 = 'x'$	$W3$ has multiple (2) inputs of types *string* and *float* from multiple component services of different service layers but no constraints	Valid Lucid definition of $W3$ with 2 local dimensions
			Figure 27	$char : input11 = 'x'$	$W1$ and $W2$ each has 1 *char* type input respectively from the user and 1 component service. Each service also has 1 constraint on its input.	Valid Lucid definitions of $W1$ and $W2$ with 1 local dimension each
			Figure 28	$float : input11 = 200.2$ $int : input12 = 300$ $int : input31 = 400$	$W2$ has multiple (2) inputs of types *string* and *boolean* from 1 component service and 2 of its own constraints and 1 adjusted constraint on its inputs	Valid Lucid definition of $W2$ with 2 local dimensions
					$W1$ has multiple (2) user inputs of types *float* and *int*, 3 of its own constraints on its inputs and 1 adjusted constraint on $W3$'s user input	Valid Lucid definition of $W1$ with 3 local dimensions

3	Evaluation context of a component service should comprise of all its local dimensions paired with either their corresponding output data members belonging to the JAVA class of their source component services or their corresponding global dimensions (if their values are provided by the user) irrespective of whether they are inputs to the given component service or features of adjusted constraints attached to its service-node.	Algorithm 11, lines 5 - 15 Computes required inputs for service processing in JAVA and constraint evaluation in LUCID. Enables storage and querying of execution results through GIPSY warehouse. Explained in Sections 2.1.4 and 4.2.	Figure 23	$float : input11 = 200.2$ $float : input41 = 300.3$ $char : input61 = 'x'$	$W3$ has multiple (2) inputs from multiple component services of different service layers but no constraints	Valid context specification with dimensions $l\_output11$ and $l\_output21$ paired with output data members of $W1$ and $W2$ respectively
			Figure 28	$float : input11 = 200.2$ $int : input12 = 300$ $int : input31 = 400$	$W1$ has multiple (2) user inputs and 1 adjusted constraint on $W3$'s user input	Valid context specification with dimensions $l\_input11$, $l\_input12$ and $l\_input31$ paired with their respective global dimensions
					$W3$ has 1 user input, multiple (2) inputs from multiple component services and 1 adjusted constraint each on its own input and $W2$'s input from another component service	Valid context specification with dimension $l\_output21$ paired with $W2$'s and $l\_output11$ and $l\_output12$ with $W1$'s output data members and $l\_input31$ with dimension $g\_input31$
4	List of constraints specified in a component service's LUCID definition should comprise of all the constraints attached to its service-node irrespective of their count, feature – own or other component service's input, data type of literal value (enclosed in single quotes if *char* and double quotes if *string* typed) and relational operator.	Algorithm 11, line 17 Enables evaluation of constraints at optimum locations	Figure 27	$char : input11 = 'x'$	$W1$ has 1 constraint with == operator on its own *char* type input	Valid constraint specification with literal value enclosed in single quotes
			Figure 28	$float : input11 = 200.2$ $int : input12 = 300$ $int : input31 = 400$	$W1$ has multiple (3) constraints with >, < and <= operators on its own *float* and *int* type inputs and 1 with >= operator on $W3$'s *int* input	Valid constraint specification with 4 constraints
					$W2$ has multiple (3) constraints with == operator on its own *boolean* and *string* type inputs	Valid constraint specification with 3 constraints and *string* type literal values enclosed in double quotes

Table 20: Composite Service Objective Lucid Translation Evaluation Summary

#	Property	Significance	Test Input			Expected/Actual Test Output
			Composite Service	Composite Service Inputs	Input Characteristics	
1	A composite service with its component services organized in a sequential structure should be correctly represented in Objective Lucid.	One of the possible organizational structures for composition plans	Figure 22	$char : input11 \; = \; 'x'$	$W1$ and $W2$ are arranged in sequence in the composition plan	Valid Objective Lucid program representing the composite service correctly and completely
2	A composite service with its component services organized in a parallel structure should be correctly represented in Objective Lucid.	One of the possible organizational structures for composition plans	Figure 23	$float : input11 \; = \; 200.2$ $float : input41 \; = \; 300.3$ $char : input61 \; = \; 'x'$	$W1 - W2 - W3$ are arranged in parallel to $W4 - W5 - W6$ in the composition plan	Valid Objective Lucid program representing the composite service correctly and completely
3	A composite service with its component services organized in a joined structure should be correctly represented in Objective Lucid.	One of the possible organizational structures for composition plans	Figure 24	$int : input11 \; = \; 100$ $float : input21 \; = \; 200.2$	Parallel component services $W1$ and $W2$ join their outputs at $W3$ in the composition plan	Valid Objective Lucid program representing the composite service correctly and completely
4	A composite service with its component services organized in a split structure should be correctly represented in Objective Lucid.	One of the possible organizational structures for composition plans	Figure 25	$boolean : input11 \; = \; true$ $string : input12 \; = \; \text{“}xyz\text{”}$	$W1$ splits its outputs between parallel component services $W2$ and $W3$ in the composition plan	Valid Objective Lucid program representing the composite service correctly and completely
5	A composite service with its component services organized in a combination of sequential, parallel, joined and split structures should be correctly represented in Objective Lucid.	One of the possible organizational structures for composition plans	Figure 26	$int : input11 \; = \; 100$ $float : input21 \; = \; 200.2$	Sequence: $W1/W2 - W3-$ $W4/W5 - W6/W7$ Parallel: $W1/W2$, $W4/W5$, $W6/W7$ Joined: $W1/W2 - W3$ Split: $W3 - W4/W5$	Valid Objective Lucid program representing the composite service correctly and completely
6	Translation of a composite service to Objective Lucid should be aborted immediately in case its input values provided by the user are found to be invalid.	Algorithm 7, line 1 Valid input values are a mandatory trigger condition for translation process	Irrelevant	$char : inputC1 \; = \; \emptyset$ $char : inputC2 \; = \; \text{“}abc\text{”}$ $int : inputI1 \; = \; \emptyset$ $int : inputI2 \; = \; \text{“}abc\text{”}$ $int : inputI3 \; = \; '-'$ $int : inputI4 \; = \; 12.34$ $float : inputF1 \; = \; \emptyset$ $float : inputF2 \; = \; \text{“}abc\text{”}$ $float : inputF3 \; = \; '-'$ $boolean : inputB1 \; = \; \emptyset$ $boolean : inputB2 \; = \; \text{“ ”}$ $boolean : inputB3 \; = \; \text{“}TRUE\text{”}$ $string : inputS1 \; = \; \emptyset$	No value provided for $inputC1$, $inputI1$, $inputF1$, $inputB1$ and $inputS1$. Too many characters for $inputC2$. Data type mismatch for $inputI2$, $inputI3$, $inputI4$, $inputF2$, $inputF3$, $inputB2$, and $inputB3$.	Inputs discarded as invalid. Appropriate error messages logged. Service translation process aborted immediately.

7	Translation process should fail in case the composite service repository specified by the user is not of an acceptable type (serialized JAVA or XML).	Repository types for which reader module is unavailable cannot be parsed	Irrelevant	Irrelevant	Composite service repository specified by the user is in JSON format	"Invalid repository file type" error logged. Service translation process aborted immediately.
8	Translation process should fail in case the composite service repository (whether serialized JAVA or XML) specified by the user does not contain the composite service to be translated.	Composite service availability is mandatory requirement for translation process	SplitCS	Irrelevant	Composite service repository specified by the user does not contain *SplitCS*	"Missing composite service" error logged. Service translation process aborted immediately.

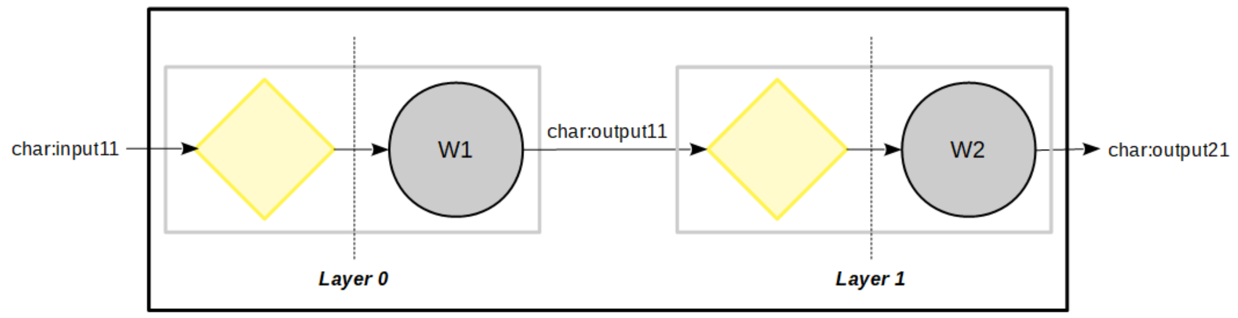


Figure 22: Unconstrained Sequentially-Organized Composite Service

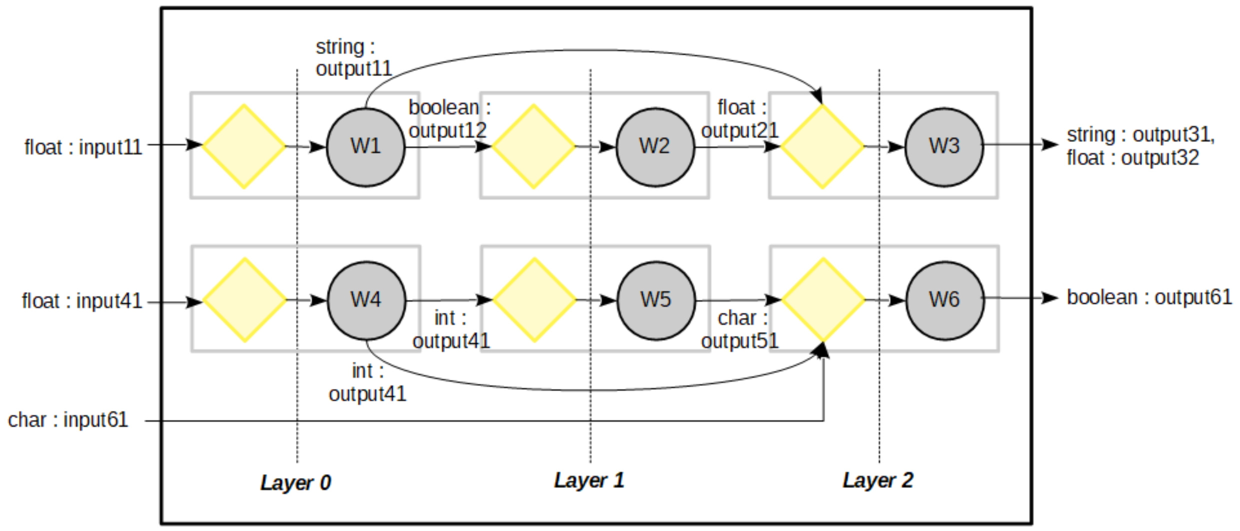


Figure 23: Unconstrained Parallelly-Organized Composite Service

The design and implementation of the translation mechanism (as described in Chapter 4) has been based upon a detailed study and analysis of both its source (i.e., layered composite services) as well as destination (i.e., Objective Lucid) sides. Working on the service composition application has helped us gain extensive knowledge of the various possible arrangements of component services in a constraint-aware plan, the relationships between them and the different adjustments in constraint locations that can occur during optimization of their evaluation points. Meanwhile, a thorough study of Objective Lucid's grammar [36, 37], program structure and execution model together with an examination of GIPSY's architecture and eductive execution approach (as presented in Sections 2.1 and 2.2) has

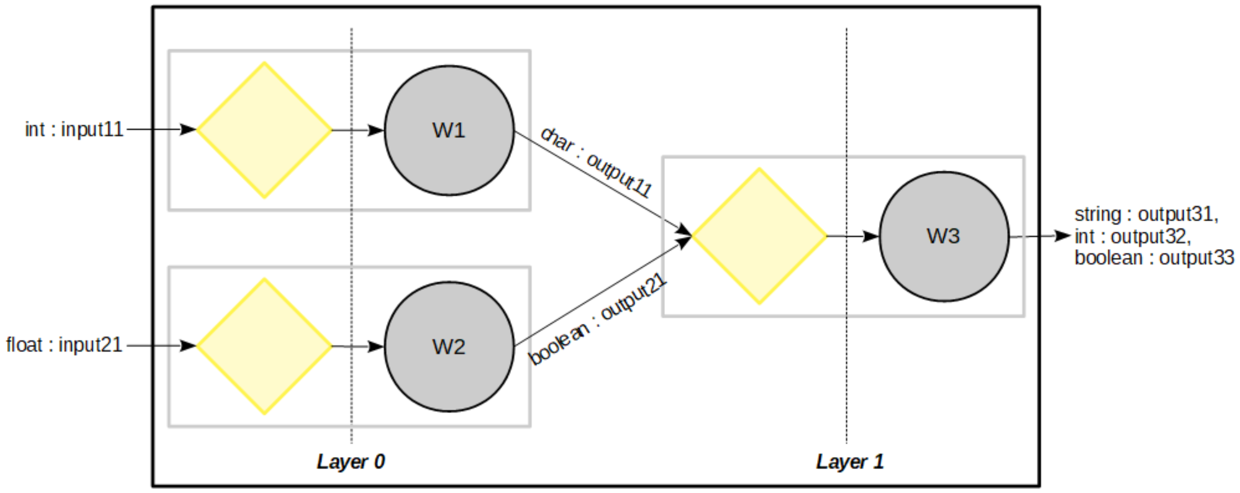


Figure 24: Unconstrained Composite Service with Joined Component-Organization

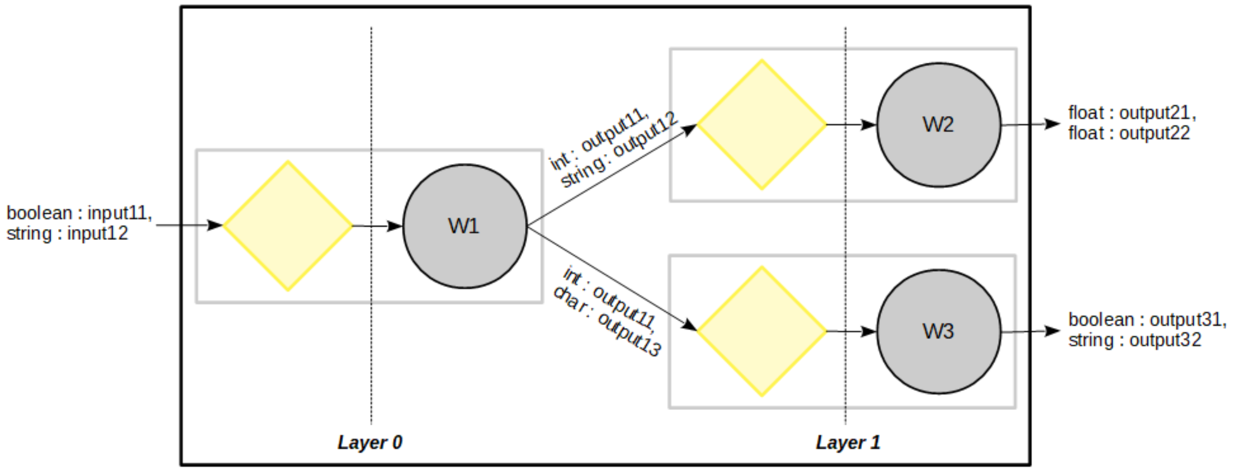


Figure 25: Unconstrained Composite Service with Split Component-Organization

significantly aided construction of clear and effective Objective Lucid representations of constraint- and context-aware composite services. The information and insights gained as a result of this entire study and analysis has been employed in designing our translation mechanism and implementing it as a flexible and maintainable software application, which, in turn, has provided us in-depth knowledge of the essential properties to be exhibited by the application, the transformations to be applied on each major and minor element of a source composite service, the exceptional scenarios to be allowed as well as the ones to be discarded

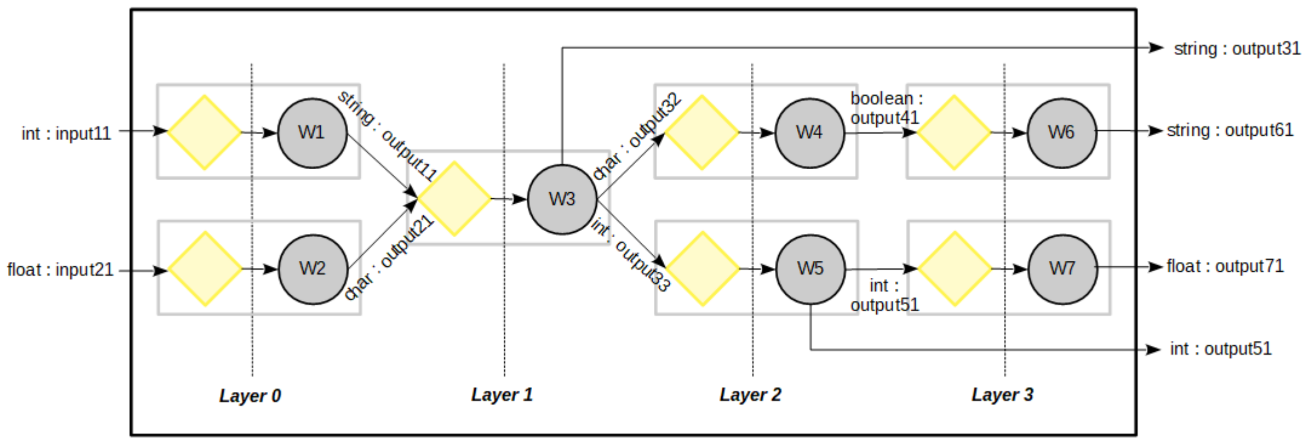


Figure 26: Unconstrained Composite Service with All Organization Structures

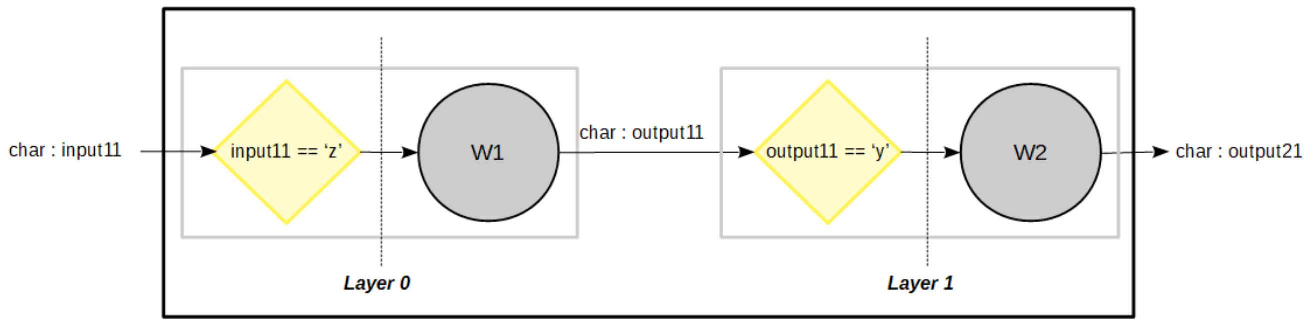


Figure 27: Constrained Sequentially-Organized Composite Service

or flagged as errors, each of which has been tested thoroughly as part of our evaluation process.

As already stated in Section 5.1, we completely understand that such a scenario-based testing technique is not an absolute proof of absence of unexpected or faulty behavior in a software application. However, designing a formal model of the Objective Lucid translation as well as its input and output entities followed by development of a proof to show that the programs generated by the translator would produce correct output for each set of composite service inputs, if and when executed, falls beyond the scope of this thesis. Taking our scope and time restrictions into consideration, our meticulous study of the composition methodology and the Lucid/GIPSY model, the comprehensive knowledge of the translator's functionalities that we gained during its design and implementation, the systematic approach

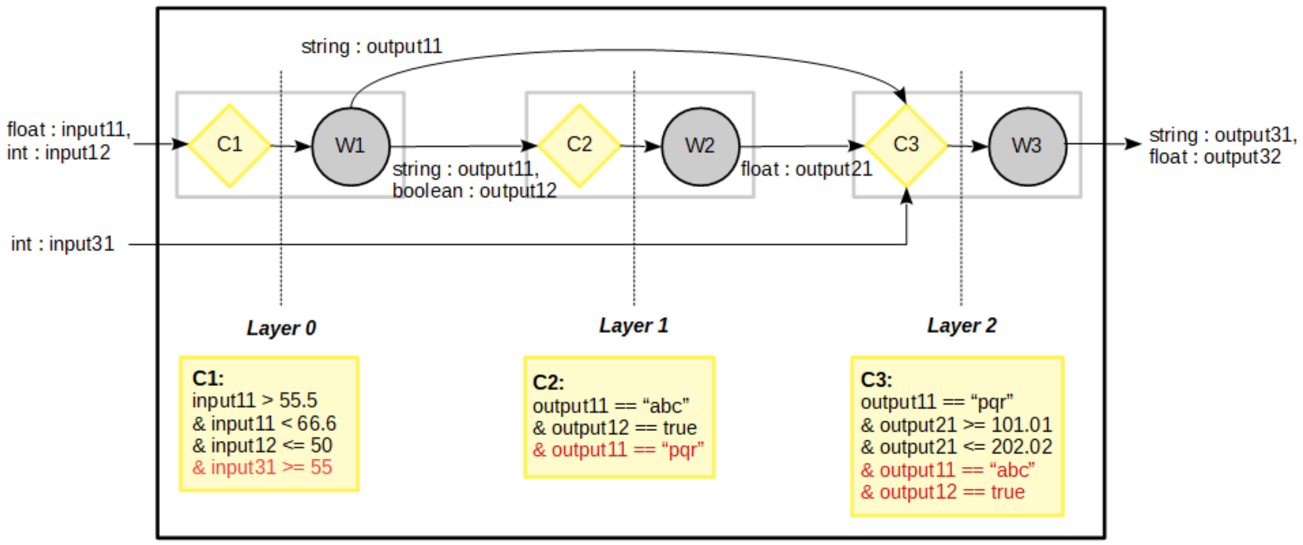


Figure 28: Sequentially-Organized Composite Service with Adjusted Constraints

that we have adopted towards designing the test cases and the successful execution of all the tests so performed allow us to conclude that we have been able to effectively evaluate our implemented solution to the best of our capabilities.

5.3 Summary

As discussed in Section 5.2, the tasks to be performed in order to fulfill the objectives of this thesis (as defined in Section 1.2) are organized as a systematic process (described in Section 1.5) aimed at verifying and validating context- and constraint-aware composite services. The first three steps of this procedure, which are concerned with the construction of layered composite services and their translation into OBJECTIVE LUCID programs, have been completed successfully as part of this research. Components of our overall verification solution that are responsible for these tasks have been designed and implemented, described in elaborate detail in Chapters 3 and 4 and thoroughly evaluated in Sections 5.1 and 5.2 to be found capable of fulfilling all their design objectives (See `https://github.com/GIPSY-dev/ServiceCompositionRepo` for the solution implementation and the tests conducted).

Unfortunately, due to unavailability of the GIPSY compiler for OBJECTIVE LUCID, the last step of the verification process concerned with the execution of the Objective Lucid representations of composite services generated by the translator framework could not be accomplished during the course of this thesis, thereby preventing us from practically demonstrating the capabilities of our verification solution as a completed system. However, we attempt to present a conceptual demonstration of the same in Sections 2.1, 2.2 and 4.2 by providing detailed explanations of Lucid's program structure, representation of context- and constraint-aware composite services as Lucid programs, means of incorporating both simulated as well as actual service functionalities in those programs, interpretation of all applicable Lucid program constructs and statements, demand generation and eductive program execution in GIPSY together with the role that Lucid's dataflow execution model and GIPSY's warehouse (owing to its demand storage and querying capabilities) play towards making the simulation and execution of context- and constraint-aware service compositions in our verification system not only possible but also time- and cost-efficient – as is intended in our thesis objectives.

To summarize, based on all the discussions presented so far and considering our time, scope and resource restrictions, in this thesis, we propose a valid simulation- and execution-based solution for the verification of context- and constraint-aware composite services, effectively describe the development of all the necessary building blocks that together constitute it and demonstrate through a meticulous evaluation approach that they satisfy all their functional requirements, thereby fulfilling the objectives of this thesis to the best of our present capabilities.

Chapter 6

Conclusion and Future Work

In this final chapter, we provide a summary of all the discussions presented so far in this thesis followed by descriptions of the limitations that we discovered in our proposed solution during the course of this research as well as the future work in which we plan to address each of them.

6.1 Conclusion

Owing to advantages such as clarity of structure, re-usability 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 (as presented in Section 2.3), we gather that the fact that no web service has a universal aptitude has been mostly overlooked in all existing research. 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 (discussed elaborately in Sections 1.1 and 2.3).

Based on the concept of context found in the intensional branch of mathematical logic (as defined in Section 2.1.1) together with the definition borrowed from [4], the execution context of a web service (whether atomic or composite) can be considered as the set of all its input parameters and the values that are 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. Additionally, since internal service constraints are, in essence, limitations placed on a service's execution context, a system meant for verifying such constraints must be able to interpret the concept of execution context. However, despite being so closely related to each other, the concepts of "service execution context" and "internal service constraints" seldom appear together in any existing research work, as per our literature review, which exhibits a significant gap in the research being conducted on web service composition (see Section 1.1 for more details).

In an attempt to address these gaps and problems related to web service composition, in this thesis, we propose the use of GIPSY (described in Section 2.2) 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 (described in Section 2.1). Therefore, as part of our proposed solution, we design and implement in JAVA an automated and extensible translator framework (described in Chapter 4) that allows modules for translating composite services into different models/languages, particularly OBJECTIVE LUCID, to be plugged-in and -out as and when required. However, before we can translate any internally-constrained compositions into Objective Lucid and simulate/execute them on GIPSY, we need to be able to generate them. To accomplish this task, we use the research conducted by Laleh et al. on composition of internally-constrained services as the base, design the plan construction algorithm missing from their composition methodology, optimize the other algorithms that they have designed and re-implement the whole process in Java as an independent, flexible and extensible software application capable of generating context- and constraint-aware composition solutions with optimally-positioned constraints for any valid composition request and set of available component services (explained in Chapter 3).

The service composition and translation methodologies employed in this thesis have been described as sets of algorithms, each of which has a specific goal to be achieved by performing a clearly-defined series of tasks. 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. All these tests have been designed based on a detailed analysis of the concepts, models and techniques involved in addition to the insights gained into the processes' behavioral intricacies while building and improving them, which has provided us with a comprehensive knowledge of the essential properties to be exhibited, each major and minor operation to be performed, the exceptional scenarios to be allowed as well as the pitfalls to be avoided by these solutions. Although we understand that such a scenario-based testing technique is not an absolute proof of absence of faulty behavior from a software application and its effectiveness is contingent upon the thoroughness with which the test cases are designed, constructing formal models of our fairly complex composition and translation solutions followed by development of mathematical proofs that establish correctness of their behavior falls beyond the scope of this thesis. Taking our scope and time restrictions under consideration, the meticulous study that we have conducted on the composition methodology and the Lucid/GIPSY model, the extensive knowledge that we have acquired on the processes' functionalities, the systematic approach that we have adopted towards designing the test cases and the successful execution of all the tests so performed allow us to conclude that we have been able to effectively evaluate our composition and translation solutions to the best of our abilities and have found them to be capable of fulfilling all their design objectives (refer to Chapter 5 for complete details of the assessment conducted).

Unfortunately, due to unavailability of the GIPSY compiler for OBJECTIVE LUCID, execution of the OBJECTIVE LUCID representations of composite services generated by our translator framework could not be accomplished during the course of this thesis, thereby preventing us from practically demonstrating the capabilities of our verification solution as a completed system. However, we attempt to present a conceptual demonstration of the same (in Sections 2.1, 2.2 and 4.2) by providing detailed explanations of representation of context- and constraint-aware composite services as LUCID programs, means of incorporating both simulated as well as actual service functionalities in those programs, interpretation of all applicable LUCID program constructs and statements and eductive program execution in GIPSY together with the role that LUCID's dataflow execution model and GIPSY's warehouse play towards making the simulation and execution of context- and constraint-aware service compositions in our verification system time- and cost-efficient – as is intended in our thesis objectives (explained further in Section 5.3).

Based on all these discussions, we conclude that, given our time, scope and resource restrictions, in this thesis, we propose a valid simulation- and execution-based solution for the verification of context- and constraint-aware composite services, effectively describe the development of all its constituent building blocks and successfully demonstrate that they satisfy all their functional requirements, thereby fulfilling the objectives of this thesis (as defined in Section 1.2) to the best of our present capabilities.

6.2 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 but have not been included, at present, due to our time and scope restrictions. Additionally, due to unavailability of the GIPSY compiler for Objective Lucid, some of the tasks planned at the beginning of this research could not be accomplished as part of this thesis. In this section, we describe all such features, enhancements and incomplete tasks that can be or are planned to be integrated into the various units of our proposed solution in the future extensions to this research.

6.2.1 Forward Expansion

The limitations found in and the future work to be undertaken for the forward expansion stage (Algorithms 2 and 3) of the service composition process have been listed below:

- 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. To further regulate the processing effort, similar conditions can be applied on other stages of the composition process as well.

- Advanced optimization techniques, such as genetic algorithms, can be integrated into the search graph construction stage for improving the quality of the composition solutions extracted from it in later stages of the process.

- According to Algorithm 3, if a service, say, $W1$, produces an output parameter, *output*1, which is an input parameter for another service, say, $W2$, then, $W1$ will be added as a predecessor to $W2$ (lines 5 - 7). This is done even if $output1$ is provided as an input parameter in the composition request (i.e., $output1 \in R.I$). However, this process, when followed in the above scenario, could result in generation of different search graphs depending on the order in which the services in the source service repository are read by the forward expansion process. For instance, considering the above example, if, instead of $W2$ being read after $W1$, $W2$ is read before $W1$ by the forward expansion algorithm, then, $W1$ would not be added as a predecessor to $W2$. In this case, $W2$ would receive $output1$ as an input from the user, and a different set of composition plans (possibly excluding $W1$) would be generated for the same composition request.

Possible Future Solution: Although this problem is out of scope and not aligned with the primary focus of this thesis, some techniques could be applied to improve the consistency of the solutions obtained in such scenarios as part of the future work. One such alternative could be to allow a service, say, $W1$, to be a predecessor to another service, say, $W2$, only if $W1$ produces an output parameter $output1$ that $W2$ takes as input and $output1$ is not included in the initial parameters ($R.I$).

6.2.2 Plan Construction

Even after the pruning and validation operations performed by the plan construction process (Algorithm 5), some of the resultant solution plans might still contain superfluous services, which could result in unnecessary expenditure of processing effort during the later stages of the verification process. For instance, consider the composition request R defined below and the set of services available for this composition listed in Table 21:

- **R.I:** $\{I_{11}, I_{21}, O_{11}, O_{21}\}$
- **R.O:** $\{O_{51}, O_{52}\}$
- **R.QoS:** $\emptyset$
- **R.C:** $\emptyset$

Table 21: Component Services Available for Resolving R

Service	**Input Parameters**	**Output Parameters**
W_1	$\{I_{11}\}$	$\{O_{11}\}$
W_2	$\{I_{21}\}$	$\{O_{21}\}$
W_3	$\{I_{11}, O_{11}\}$	$\{O_{31}\}$
W_4	$\{I_{21}, O_{21}\}$	$\{O_{41}\}$
W_5	$\{O_{31}\}$	$\{O_{51}, O_{52}\}$
W_6	$\{O_{41}\}$	$\{O_{51}, O_{52}\}$

When the plan construction process is executed for the given request and services, seven composition plans, as depicted in Figure 29, are generated as possible solutions. Since, our current model does not take into account the Quality of Service features, strictly in terms of the number of component services, plans 1 and 2 are the optimum solutions. However, five other plans are also generated besides them, which contain more component services and are likely to consume more resources for further processing as well as for simulation/execution.

Possible Future Solution: Although such additional plans offer alternative solutions in case the optimum ones fail at any point and could even turn out to be better solutions if all quality features are considered, further optimization might still be desirable. To accomplish that, another stage dedicated to optimizing the services composed into a solution plan can be added to the service composition process as part of future updates to the existing design.

6.2.3 Constraint-aware Plan Construction

More complex solutions can be employed in future to differentiate between internal constraints and eliminate the duplicates from a service-node as part of the constraint-aware plan construction stage (Algorithm 6). This would reduce the number of constraints to be evaluated within and, consequently, the execution-time required for resultant constraint-aware composition plans.

Intuitively, a simple differentiation technique could be to respectively compare the features, operators and literal values of the constraints in question and discard the ones that match another constraint in all three elements. However, this approach does not always

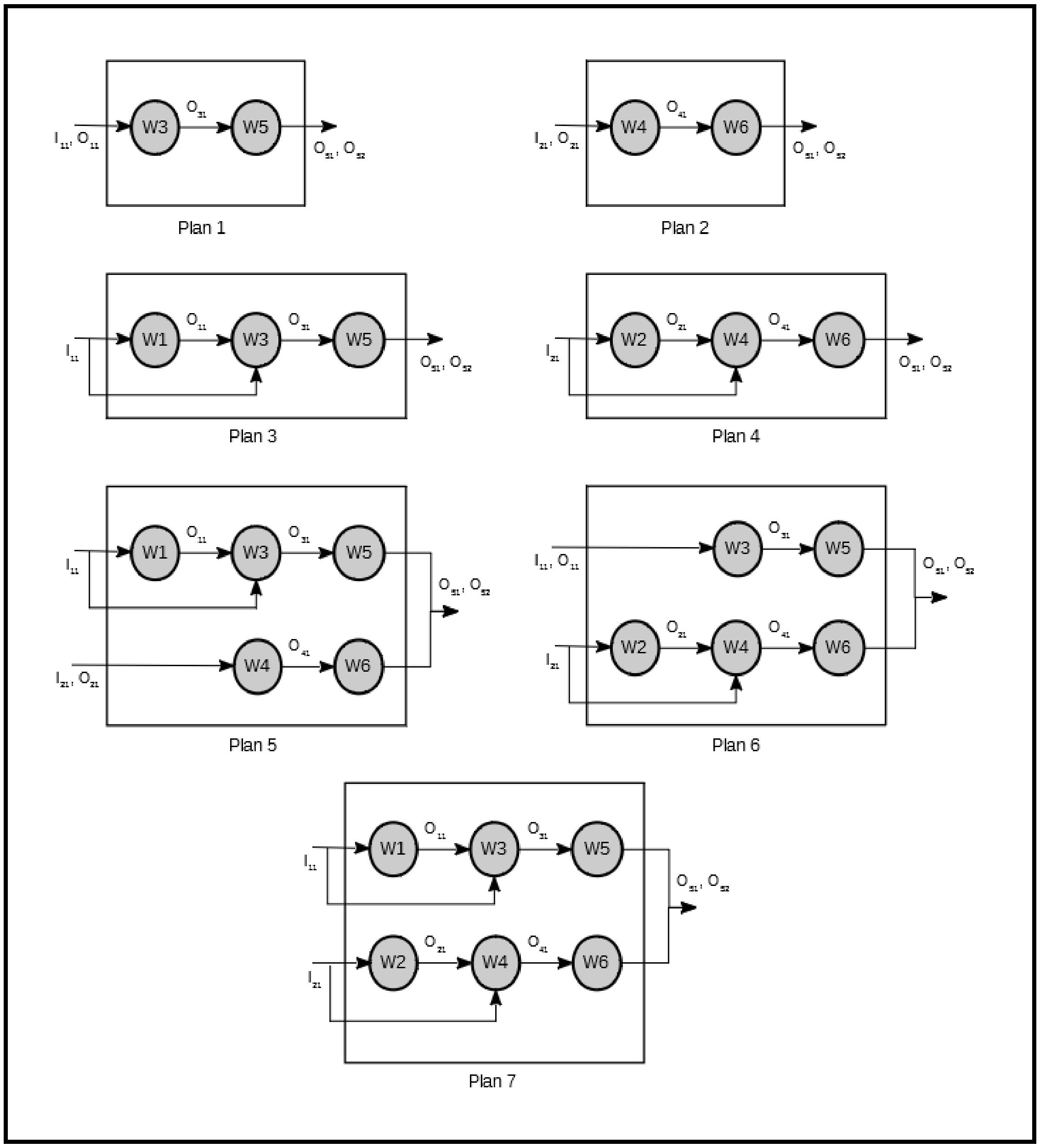


Figure 29: Solution Plans Generated for R

recognize the duplicates. For example, by this technique, constraint "*int* : $Price < 100$" is different from "*int* : $Price <= 99$" whereas mathematically both the constraints denote the same range of values for *Price*. Due to such complexities involved and considering the scope of the present solution, no technique for differentiating between constraints (other than eliminating duplicate Java objects) could be proposed in this thesis.

6.2.4 Service Composition

The limitations found in and the future work to be undertaken for the service composition process in general have been listed below:

- Although we allow service requesters to specify their constraints and expected QoS features as part of a composition request for the sake of completeness (see Definition 3), we do not implement any mechanisms as of now that would enable requester constraints or QoS features to be included in a solution plan (see Section 3.5). Also, while Algorithm 2 (forward expansion) does include a *CheckRequesterConstraints* statement (line 7) that should trigger verification of requester constraints, currently, it acts only as a placeholder for a more elaborate solution, designing which is out of scope of this thesis. However, we plan to incorporate QoS features and verify requester constraints placed on service compositions in the future works extending from this thesis.

- In our current implementation, the operators that can be used in constraints include only $<$, $>$, $=$, $<=$ and $>=$ while the QoS features that can be specified in composition requests include only *COST*, *RESPONSE_TIME*, *RELIABILITY* and *AVAILABILITY* [5]. Additionally, *int*, *char*, *float*, *string* and *boolean* are the only data types being handled for service input and output parameters by our present solution (see Section 3.5.2). In future extensions, more operators and QoS features can be added to the *Operator* [51] and *QualityOfService* enumerations respectively and means to validate and process them can be added to our implementation while extending it to handle additional data types for service inputs and outputs.

- Currently, a service parameter is represented as a *String* JAVA object with two parts – data type and name – separated by a colon in our implementation (see Section 3.5.2). This makes the parsing of service parameters untidy. To make the process cleaner, a new data structure, such as a *Parameter* class with *paramDataType* and *paramName* data members, can be defined to better represent and process service inputs, outputs, effects and constraint features in the future.

- The storage and reuse of layered composite services (described in Section 3.5.4.6) is, at present, restricted to serialized JAVA object repositories in our solution. This functionality can be extended in future to other repository formats, such as, XML, JSON, WSDL, etc., by designing proper representations and developing relevant parsers and writers for them.

- The research conducted by Laleh et al. includes an algorithm for merging all the solutions generated for a given composition request into a single package so that, in case one solution plan fails during run-time constraint verification, other alternative plans from the package can be used for obtaining the desired output, thus broadening the contextual range over which the generated solution can operate [4, 5]. While we have restricted our current implementation to the composition and verification of a single plan at a time in order to avoid complicating our prototype system at such an early stage, we plan to incorporate construction of solution packages into our system as part of a future work.

6.2.5 Composite Service Translation/Simulation/Execution

The limitations found in and the future work to be undertaken for the translation of composite services into OBJECTIVE LUCID followed by their simulation/execution on GIPSY have been listed below:

- As per the current design of the OBJECTIVE LUCID translator, body of the *process* member method of a component service's JAVA class representation comprises of nothing but simple placeholder statements that assign dummy values to output data members based on their data type to ensure that our current solution passes the basic sanity checks. For more advanced testing and verification, as part of future extensions, we plan to include an additional component in the atomic service model described by Laleh et al. (Definition 1) that would specify the implementation (whether simulated, actual or linked) of the service and could be extracted by the Objective Lucid translator module so as to replace the current placeholder implementation of the corresponding service's *process* method.

- While our composition application allows layered 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. As future work, 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.

- As mentioned in Section 6.2.4, we plan to incorporate construction of solution packages into our composition application in future. Once this construction is accomplished, the translator framework can also be updated to allow translation of the packages so generated into various target models/languages.

- Due to unavailability of the GIPSY compiler for Objective Lucid, we have not been able to provide a practical demonstration of the capabilities of our verification solution as a completed system as part of this thesis. Moreover, due to our time and scope limitations, we have been unable to provide a formal/mathematical proof to establish the correctness of our composition and translation solutions. However, future research works based on this thesis can be and are planned to be dedicated to accomplishing these evaluation tasks.

References

[1] R. Jayakumar and S. K. Narula, “Distributed system design, course notes for COMP6231, summer 2017.” [online], June 2017.

[2] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair, *Distributed Systems: Concepts and Design*. Addison-Wesley, 5 ed., 2012. ISBN: 978-0-13-214301-1.

[3] P. Wang, Z. Ding, C. Jiang, and M. Zhou, “Constraint-aware approach to web service composition,” *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, vol. 44, pp. 770–784, June 2014.

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

[5] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, “Constraint verification failure recovery in web service composition,” *Future Generation Computer Systems*, vol. 89, pp. 387 – 401, 2018. http://www.sciencedirect.com/science/article/pii/S0167739X17320629.

[6] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, “Predictive failure recovery in constraint-aware web service composition,” in *Proceedings of the 7th International Conference on Cloud Computing and Services Science - Volume 1: CLOSER*, pp. 241–252, INSTICC, SciTePress, 2017.

[7] T. Laleh, J. Paquet, S. Mokhov, and Y. Yan, “Constraint adaptation in web service composition,” in *2017 IEEE International Conference on Services Computing (SCC)*, pp. 156–163, June 2017.

[8] T. Laleh, J. Paquet, S. A. Mokhov, and Y. Yan, “Efficient constraint verification in service composition design and execution (short paper),” in *CoopIS*, pp. 445–455, Springer, 2016.

[9] R. Aggarwal, K. Verma, J. Miller, and W. Milnor, “Constraint driven web service composition in METEOR-S,” in *IEEE International Conference on Services Computing, 2004. (SCC 2004). Proceedings. 2004*, pp. 23–30, Sept 2004.

[10] G. Chafle, K. Dasgupta, A. Kumar, S. Mittal, and B. Srivastava, “Adaptation in web service composition and execution,” in *2006 IEEE International Conference on Web Services (ICWS’06)*, pp. 549–557, Sept 2006.

[11] S. Youcef, M. U. Bhatti, L. Mokdad, and V. Monfort, “Simulation-based response-time analysis of composite web services,” in *2006 IEEE International Multitopic Conference*, pp. 349–354, Dec 2006.

[12] C. Zhu and Y. Du, “Application of logical petri nets in web service composition,” in *2010 IEEE International Conference on Mechatronics and Automation*, pp. 913–918, Aug 2010.

[13] M. Chen, T. H. Tan, J. Sun, Y. Liu, and J. S. Dong, “VeriWS: A tool for verification of combined functional and non-functional requirements of web service composition,” in *Companion Proceedings of the 36th International Conference on Software Engineering*, ICSE Companion 2014, (New York, NY, USA), pp. 564–567, ACM, 2014.

[14] K. T. Huynh, T. T. Quan, and T. H. Bui, “Fast and formalized: Heuristics-based on-the-fly web service composition and verification,” in *2015 2nd National Foundation for Science and Technology Development Conference on Information and Computer Science (NICS)*, pp. 174–179, Sept 2015.

[15] V. Shkarupylo, “A simulation-driven approach for composite web services validation,” in *Central European Conference on Information and Intelligent Systems*, p. 227, Faculty of Organization and Informatics Varazdin, Sept 2016.

[16] S. Narayanan and S. A. McIlraith, “Simulation, verification and automated composition of web services,” in *Proceedings of the 11th International Conference on World Wide Web*, WWW ’02, (New York, NY, USA), pp. 77–88, ACM, 2002.

[17] S. Narayanan and S. McIlraith, “Analysis and simulation of web services,” *Computer Networks*, vol. 42, no. 5, pp. 675 – 693, 2003. The Semantic Web: an evolution for a revolution.

[18] X. Wang and S. Yu, “A novel method for verification of composite web services,” in *2015 2nd International Conference on Information Science and Control Engineering*, pp. 37–40, April 2015.

[19] C. Dechsupa, W. Vatanawood, and A. Thongtak, “Formal verification of web service orchestration using colored petri net,” in *Proceedings of the International MultiConference of Engineers and Computer Scientists*, vol. 1, March 2016.

[20] X. Fu, T. Bultan, and J. Su, “Analysis of interacting BPEL web services,” in *Proceedings of the 13th international conference on World Wide Web*, pp. 621–630, ACM, 2004.

[21] A. Khodadadi, “Collection and classification of services and their context,” Master’s thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Sept. 2015.

[22] J. Paquet and P. G. Kropf, “The GIPSY architecture,” in *Proceedings of Distributed Computing on the Web* (P. G. Kropf, G. Babin, J. Plaice, and H. Unger, eds.), vol. 1830 of *Lecture Notes in Computer Science*, pp. 144–153, Springer Berlin Heidelberg, 2000.

[23] J. Paquet and A. H. Wu, “GIPSY – a platform for the investigation on intensional programming languages,” in *Proceedings of the 2005 International Conference on*

Programming Languages and Compilers (PLC 2005), pp. 8–14, CSREA Press, June 2005.

[24] J. Paquet, "Distributed eductive execution of hybrid intensional programs," in *Proceedings of the 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09)*, pp. 218–224, IEEE Computer Society, July 2009.

[25] E. A. Ashcroft, A. A. Faustini, R. Jagannathan, and W. W. Wadge, *Multidimensional Programming*. London: Oxford University Press, Feb. 1995. ISBN: 978-0195075977.

[26] W. W. Wadge and E. A. Ashcroft, *Lucid, the Dataflow Programming Language*. London: Academic Press, 1985.

[27] P. Rondogiannis and W. W. Wadge, "Intensional programming languages," in *Proceedings of the First Panhellenic Conference on New Information Technologies (NIT'98), Athens, Greece*, pp. 85–94, 1998.

[28] B. Han, "Towards a multi-tier runtime system for GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, 2010.

[29] J. Paquet, *Scientific Intensional Programming*. PhD thesis, Department of Computer Science, Quebec City, Canada, 1999.

[30] J. Cheng, C. Liu, M. Zhou, Q. Zeng, and A. Yla-Jaaski, "Automatic composition of semantic web services based on fuzzy predicate petri nets," *IEEE Transactions on Automation Science and Engineering*, vol. 12, pp. 680–689, April 2015.

[31] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber, "The petri net markup language: Concepts, technology, and tools," in *Applications and Theory of Petri Nets 2003* (W. M. P. van der Aalst and E. Best, eds.), (Berlin, Heidelberg), pp. 483–505, Springer Berlin Heidelberg, 2003.

[32] "Pnml.org - PNML reference site." http://www.pnml.org/, viewed in August 2018.

[33] S. Juan and W. Hao, "Performance analysis for web service composition based on queueing petri net," in *Software Engineering and Service Science (ICSESS), 2012 IEEE 3rd International Conference on*, pp. 501–504, IEEE, 2012.

[34] R. Jagannathan and C. Dodd, "GLU programmer's guide," tech. rep., SRI International, Menlo Park, California, 1996.

[35] R. Jagannathan, C. Dodd, and I. Agi, "GLU: A high-level system for granular data-parallel programming," in *Concurrency: Practice and Experience*, vol. 1, pp. 63–83, 1997.

[36] S. A. Mokhov, "Towards hybrid intensional programming with JLucid, Objective Lucid, and General Imperative Compiler Framework in the GIPSY," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Oct. 2005. ISBN 0494102934; online at http://arxiv.org/abs/0907.2640.

[37] S. Mokhov and J. Paquet, "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)*, pp. 22–28, CSREA Press, June 2005.

[38] Sun Microsystems, Inc., "The Java web services tutorial (for Java Web Services Developer's Pack, v2.0)." [online], Feb. 2006. http://download.oracle.com/docs/cd/E17802_01/webservices/webservices/docs/2.0/tutorial/doc/.

[39] F. Siala, I. Ait-Sadoune, and K. Ghedira, "A multi-agent based approach for composite web services simulation," in *International Conference on Model and Data Engineering*, pp. 65–76, Springer, 2014.

[40] AT&T Labs Research and Various Contributors, "Graphviz – graph visualization software." [online], 1996–2012. http://www.graphviz.org/.

[41] E. R. Gansner, E. Koutsofios, and S. North, *Drawing graphs with dot*, January 2015. https://graphviz.gitlab.io/_pages/pdf/dotguide.pdf.

[42] D. Nagamouttou, I. Egambaram, M. Krishnan, and P. Narasingam, "A verification strategy for web services composition using enhanced stacked automata model," *SpringerPlus*, vol. 4, no. 1, p. 98, 2015.

[43] N. Adadi, M. Berrada, D. Chenouni, and B. Bounabat, "Modeling and simulation of web services composition based on MARDS model," in *Intelligent Systems: Theories and Applications (SITA), 2015 10th International Conference on*, pp. 1–6, IEEE, 2015.

[44] S. Majithia, M. Shields, I. Taylor, and I. Wang, "Triana: A graphical web service composition and execution toolkit," in *Web Services, 2004. Proceedings. IEEE International Conference on*, pp. 514–521, IEEE, 2004.

[45] K. J. Turner, "Representing and analysing composed web services using CRESS," *Journal of network and computer applications*, vol. 30, no. 2, pp. 541–562, 2007.

[46] S. Chandrasekaran, G. Silver, J. A. Miller, J. Cardoso, and A. P. Sheth, "XML-based modeling and simulation: web service technologies and their synergy with simulation," in *Proceedings of the 34th conference on Winter simulation: exploring new frontiers*, pp. 606–615, Winter Simulation Conference, 2002.

[47] G. A. Silver, A. Maduko, R. Jafri, J. A. Miller, and A. P. Sheth, "Modeling and simulation of quality of service for composite web services," in *Proceedings of the 7th World Multiconference on Systemics, Cybernetics and Informatics (SCI'03)*, vol. 1, pp. 420–425, 2003.

[48] S. Chandrasekaran, J. A. Miller, G. S. Silver, B. Arpinar, and A. P. Sheth, "Composition, performance analysis and simulation of web services," 2002.

[49] S. Chandrasekaran, J. A. Miller, G. S. Silver, B. Arpinar, and A. P. Sheth, "Performance analysis and simulation of composite web services," *Electronic Markets*, vol. 13, no. 2, pp. 120–132, 2003.

[50] S. Software, "SoapUI Projects." Published electronically, https://www.soapui.org/soapui-projects/soapui-projects.html. [accessed 10-May-2019], 2019.

[51] A. Simard, "A framework for interoperability across heterogeneous service description models," Master's thesis, Department of Computer Science and Software Engineering, Concordia University, Montreal, Canada, Mar. 2019.