

Notre Dame University
Faculty of Natural and Applied Sciences
Department of Computer Science

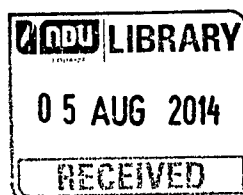
On-the-fly Algorithm for the Service Composition Problem

**A Thesis Submitted in Partial Fulfillment of the
Requirements for the Joint Degree of the Master of
Sciences in Computer Science – Computer Information Systems**

By

Ceaser Younes

NDU-Lebanon
2014



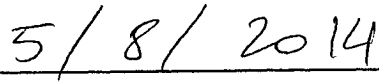
Thesis Release Form

I, Caesar Younes, authorize Notre Dame University-Louaize to supply copies of my thesis to libraries or individuals on request.

I, _____, do not authorize Notre Dame University-Louaize to supply copies of my thesis to libraries or individuals on request.

A handwritten signature in black ink, consisting of several overlapping loops and a horizontal stroke, positioned above a horizontal line.

Signature

A handwritten date in black ink, "5/8/2014", positioned above a horizontal line.

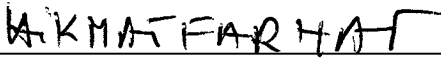
Date

On-the-fly Algorithm for the Service Composition Problem

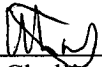
By

Ceaser Younes

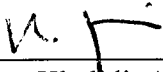
Approved by:



Hikmat Farhat: Associate Professor of Computer Science
Advisor



Khalil Challita: Assistant Professor of Computer Science
Member of Committee



Khaldoun Khalidi: Associate Professor of Computer Science
Member of Committee

Date of Thesis Defense: June 25th, 2014

Abstract

Web services are a form of middleware to exchange information between systems over a network. With the advent of the age of mobile devices and their diversity, the importance of web services became all the more apparent. Every web service grants certain functionality and can utilize other web services to gain more functionality and more robustness. Such a service that combines the functionality of other services is called a composite service and the process of designing such a service is called a composition. This thesis studies an on-the-fly algorithm, that efficiently checks for the possibility of matching a target composition from a community of services. The algorithm proposes that a match for the target service can be found on-the-fly by visiting a select number of service states instead of having to parse the entire state space. Furthermore, On-the-fly can be paired with some heuristics or business rules for faster implementation and higher quality of service. The correctness of the algorithm was proven and the complexity was shown to be optimal. The implementation on test cases is very promising.

Table of Contents

Abstract	2
Table of Contents	3
List of Figures	4
List of Abbreviations	5
Acknowledgements.....	6
Chapter 1: Introduction, Problem Definition, and Approach.....	7
1.1 Introduction to Web Services	7
1.2 Composition, Orchestration, and BPEL.....	11
1.3 Service Composition	17
1.4 Approach and Main Results	18
1.5 Thesis Organization.....	18
Chapter 2: Model and On-the-Fly Algorithm	19
2.1 Available Services.....	19
2.2 Target Service.....	20
2.3 Service Composition	23
2.4 On-the-fly Algorithm.....	27
2.5 Correctness and Complexity	33
Chapter 3: Implementation	36
3.1 Basic Data Structures	36
3.2 Implementation	38
3.3 Fixed Point Algorithm.....	43
3.4 Simulation Results and Interpretation.....	46
Chapter 4: Conclusions	50
4.1 Summary of the main results.....	50
References	51

List of Figures

Figure 1.1. Web Services Business Architecture	9
Figure 1.2. Business case graph	11
Figure 1.3 Orchestration Diagram	13
Figure 1.4 Scope of the	14
Figure 1.5 Orchestration in BPEL	15
Figure 2.1 Target Service	21
Figure 2.2 User Authentication	22
Figure 2.3 Sales	22
Figure 2.4 Accounting	22
Figure 2.5 State space	26
Figure 3.1 Simple Test Case	40
Figure 3.3 On-the-fly result	46
Figure 3.4 Fixed Point Result	47
Figure 3.5 Running time comparison for small community	48
Figure 3.6 Running time vs number of services	49

List of Abbreviations

W3C: World Wide Web Consortium

LAN: Local Area Network

XML: eXtensible Markup Language

SOAP: Simple Object Access Protocol

WSDL: Web Services Description Language

UDDI: Universal Description, Discovery and Integration

BPEL: Business Process Execution Language

OASIS: Organization for the Advancement of Structured Information Standards

JVM: Java Virtual Machine

JIT: Just In Time Compilation

Acknowledgements

The masters thesis is the culmination of three years of hard work and the result of which one person takes a degree. Several people are due credit and yet none is given. With the completion of my masters I would like to extend a warm thank you to these people.

To Dr. Hikmat Farhat, my thesis advisor and mentor, thank you for always being patient and always being there to lend a helping hand.

To Dr. Hoda Maalouf, Chairperson of the department, thank you for your guidance throughout the years it would not have been possible without you.

To my professors, thank you for all the knowledge you've given. Sir Isaac Newton once famously said "If I have seen further it is by standing on the shoulders of giants" and I can only extend the same kind words to you.

To my family, girlfriend, and God thank you for your continued support and belief in me.

Chapter 1: Introduction, Problem Definition, and Approach

Technology is converging towards networking, connectivity, and the Internet. Software as a service, service oriented applications, and cloud services are all words that describe the spirit of the times in the domain of information technology. Web services played a major role in getting us here and, from their inception, were developed for satisfying such goals. Web services are an evolution of middleware in an attempt to make use of the connectivity provided by the Internet. From a business stand point they were meant to facilitate Enterprise Application Integration or EAI for short. We will discuss a brief history of how and why they were created, bringing us to the standards followed and what the On-the-fly algorithm aims to improve.

1.1 Introduction to Web Services

1.1.1 The need for Web Services

All companies of all sizes are comprised of several departments. Each department has its own business flow implemented in the software it uses to accomplish its tasks. As companies grow they spread to bigger or sometimes geographically numerous locations. The term Enterprise Application Integration refers to integrating the existing systems within the enterprise in order to share or replicate data and to execute business processes that require interaction between several departments in several branches.

With data being decentralized and distributed there was only one obvious solution; middleware. There had to be some software that would act as a mediary especially when retrieving data from or writing to legacy systems while maintaining the business process. Web services were used as a form of middleware because they added an extra layer of abstraction. This simplified the design of the clients by accessing the underlying resources and fetching the data from the source readily. Effectively the web services acted as wrappers that provided an interface to communicate with any system. This leveraged the power of the internet and made a company's resources available beyond the reach of its Local Area Network (LAN).

For automation and interoperability among heterogeneous systems to work there were some issues that had to be addressed. There had to be a generic interface for calling the service and a standard syntax in which the information returned could be understood.

Before going into the standards and modus operandi of web services we must establish what a web service is. According to the World Wide Web Consortium (W3C) a Web Service is “a software application identified by a URI, whose interfaces and bindings are capable of being defined, described, and discovered as XML artifacts. A Web service supports direct interactions with other software agents using XML-based messages exchanged via Internet-based protocols”. The definition mentions three important properties of Web Services; they should be capable of being defined, described, and discovered. These three properties are what make Web Services superior to other middleware. They indicate that Web Services are described and advertised thus giving them the possibility to be publicly advertised and used to create more complex services and distributed applications.

For interaction to occur among different companies and different systems within a company, standardization was a must. Standardization had to occur at the level of architecture and protocols as well as at the level of syntax and language. As with the web itself, standardization is the key to progress but that does not mean there is one approach or specification for every aspect. Web services are fragmented with different specifications developed by different companies. As the different standards get adopted by the big companies the number of standards will get smaller and sometimes converge to one.

1.1.2 Web Service Specifications

There are many views for web services architecture but the most popular one currently in use was proposed by IBM which is comprised of three elements; service requestor, service provider, and service registry. It follows the most commonly used specifications which are SOAP, WSDL, and UDDI. First we have a service requester who is the potential user of a service. He would be looking for services published by a service provider, the entity that implements the service and offers to carry it out on behalf of the requester. The services would be published in a service

registry, a place where available services are listed and which allows providers to advertise their services and requesters to query for services.

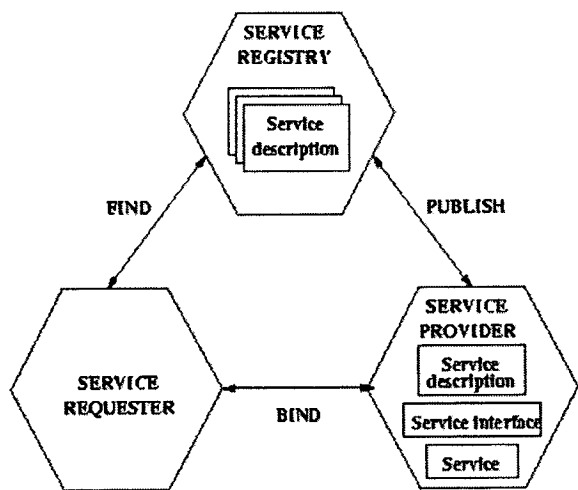


Figure 1.1. Web Services Business Architecture

Web services are more of a field of study rather than a set of specifications. There are many aspects to them with competing specifications for each aspect.

First and foremost a common language is needed as the basis for specifying all the languages necessary to describe the different aspects of a service. XML is used for this purpose, both because it is a widely adopted and commonly accepted standard and because its syntax is flexible enough to enable the definition of service description languages and protocols.

The Simple Object Access Protocol (SOAP) was initiated by W3C in 1999. SOAP covers the message format for one-way communication describing how a message can be packed into an XML document. It includes a set of rules that must be followed when processing a SOAP message and a simple classification of the entities involved in processing a SOAP message. It also specifies what parts of the messages should be read by whom and how to react in case the content is not understood [2].

WSDL discusses how to describe the different parts that comprise a Web service. It encompasses an abstract description containing the type system used to describe the messages (based on XML Schema), the messages involved in invoking the service, the individual operations composed of

different message exchange patterns, and an interface that groups the operations that constitute an abstract service.

It also explicitly defines a binding the interface to a transport protocol, the endpoint or network address of the binding, and a service as a collection of all bindings of the same interface [2].

The UDDI specification is probably the one that has evolved the most from all specifications we have seen so far. Originally, UDDI was conceived as a “Universal Business Registry” similar to search engines (e.g., Google) which will be used as the main mechanism to find electronic services provided by companies worldwide. This triggered a significant amount of activity around very advanced and complex scenarios like Semantic Web, dynamic binding to partners, runtime/automatic partner selection and others. Nowadays UDDI is far more pragmatic and recognizes the realities of B2B interactions: it presents itself as the “infrastructure for Web services”, meaning the same role as a name and directory service (i.e., binder in RPC) but applied to Web services and mostly used in constrained environments (internally within a company or among a predefined set of business partners) [2].

1.2 Composition, Orchestration, and BPEL

1.2.1 Web service composition

To put the problem into perspective we will take a business case. Consider a store that sells some goods and a certain item is running low in stock or a new item needs to be added to the store. The standard procedure would be to contact multiple suppliers and request quotes, select the supplier that offers the best deal, request approval for a purchase amount, and finally order the goods from the supplier.

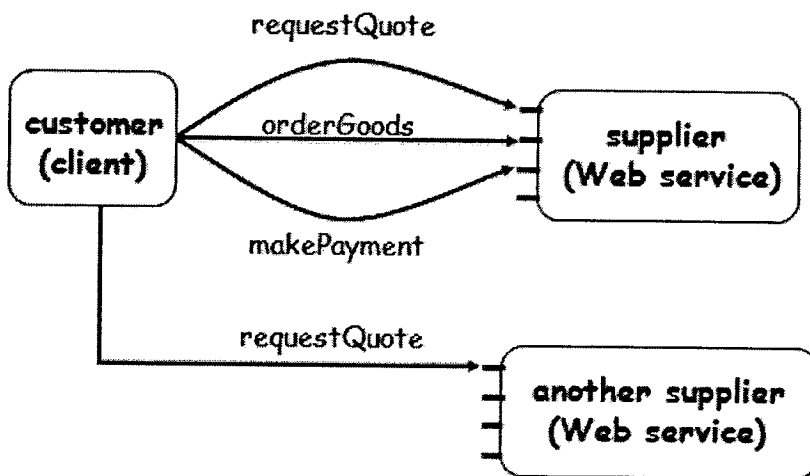


Figure 1.2. Business case graph

This is essentially a business workflow composed of several operations. For web services to achieve the above workflow the operations of several web services will have to be combined giving us a web service composition. An important thing to note, however, is that a service composition is not equal to the sum of its parts. In enterprise applications a composition of components would be to add the components to the application and utilize the functionality of each. Web services are interfaces that define an input, output, and action performed. The composition would specify what services to call, in what order, and how to handle error exceptions. The components themselves remain separate and external to the application. Composition started out as hard-coding in traditional languages like Java or C#. This was the approach implemented by previous middleware and naturally web services were modeled to

what was already familiar. Composite web services were used as a way to bridge heterogeneous middleware platforms. The problem with this approach is that it makes a composition essentially the same as a non-composite service. Thus composition as an activity is not supported, abstraction is not possible, and the infrastructure is static and restricting [2].

1.2.2 Orchestration

An orchestration defines the sequence and conditions in which one Web service invokes other Web services in order to achieve some useful function. I.e., an orchestration is the pattern of interactions that a Web service agent must follow in order to achieve its goal. In orchestration, which is usually used in private business processes, a central process (which can be another Web service) takes control of the involved Web services and coordinates the execution of different operations on the Web services involved in the operation. The involved Web services do not "know" (and do not need to know) that they are involved in a composition process and that they are taking part in a higher-level business process. Only the central coordinator of the orchestration is aware of this goal, so the orchestration is centralized with explicit definitions of operations and the order of invocation of Web services [2].

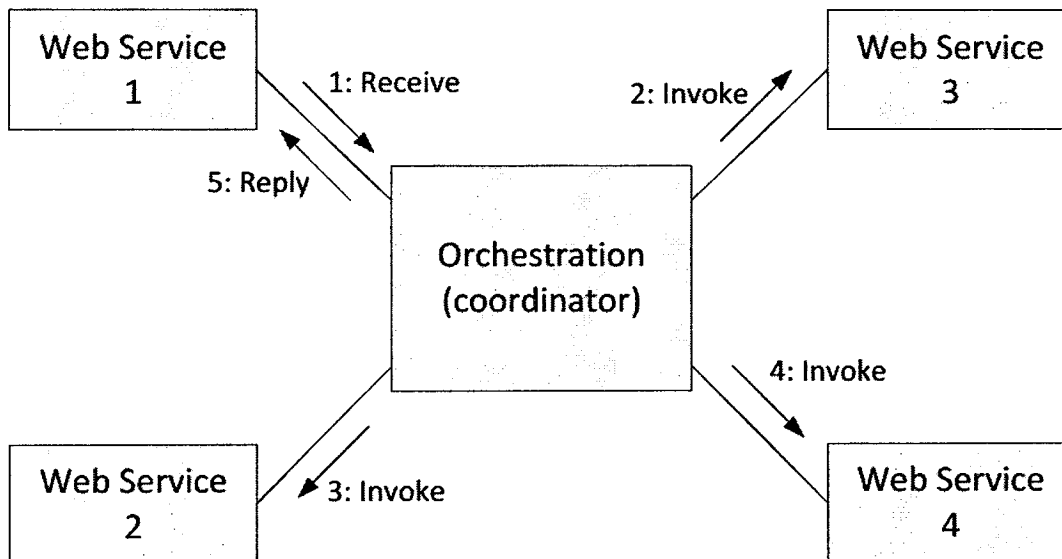


Figure 1.3 Orchestration Diagram

The result of an orchestration is a composition. Coincidentally it seems the pun in the names was intended. An orchestra is a group of musicians each playing his own instrument. An orchestration in music terms is arranging these musical instruments to play a piece together harmoniously. Different instruments will be playing different notes but they come together to deliver a bigger piece. This musical piece is called a composition and it is lead by an orchestra conductor; an orchestrator. The instruments are the web services, conductor is the orchestrator, and the orchestration or the composition has the same name in both web services and music. Similar to how an orchestra conductor maintains the lead an orchestrator in orchestration is the central authority on how the process is run.

1.2.3 BPEL

Business Process Execution Language (BPEL) defines a notation for specifying business process behavior based on Web services. It has been proposed as a way to model and program the composition by combining services whose interfaces are specified in WSDL. It does not directly deal with an implementation of the language but only with the semantics of the primitives it provides. The emphasis is on the interoperability between systems rather than portability of specifications.

It was initially proposed by IBM, Microsoft, and BEA and is now being standardized by OASIS. The language reflects in many ways the characteristics of earlier attempts, which are WSFL and XLANG, proposed by the same vendors.

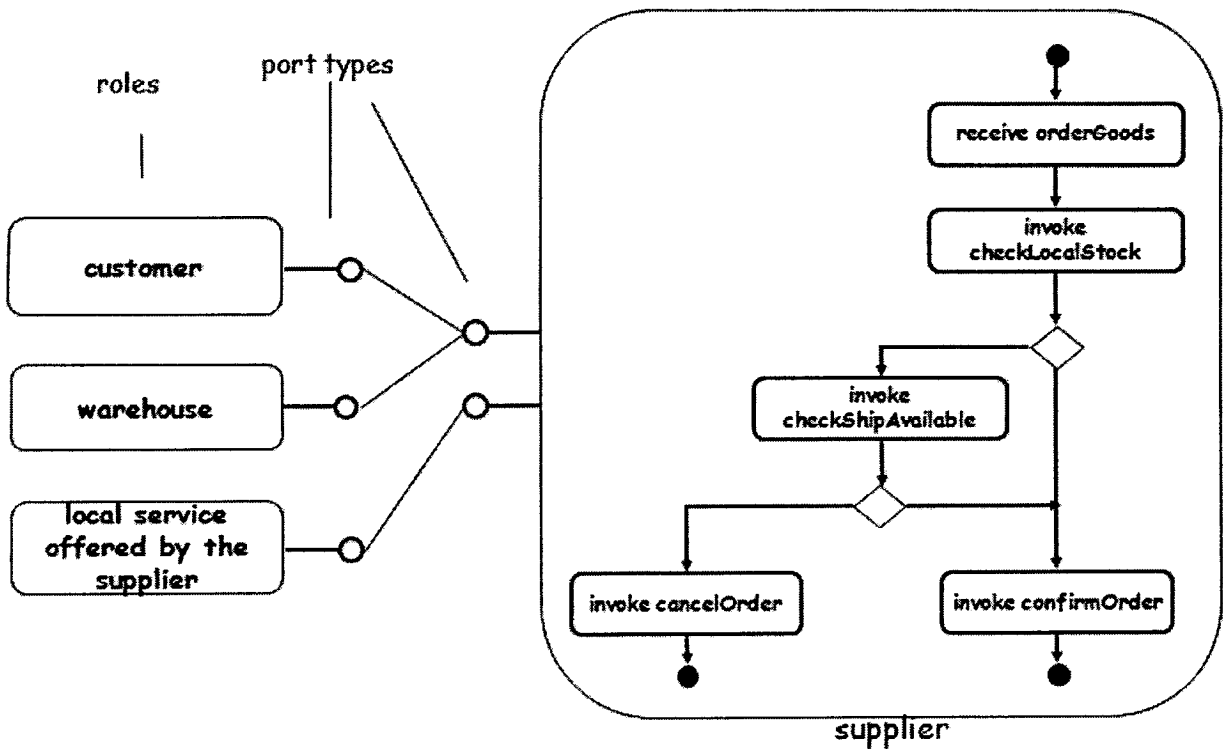


Figure 1.4 Scope of the BPEL

In a nutshell, BPEL specifications are XML documents that define the following aspects of a process:

- The different roles that take part in the message exchanges with the process
- The port types that must be supported by the different roles and by the process itself
- The orchestration and the other different aspects that are part of a process definition
- Correlation information, defining how messages can be routed to the correct composition instances

As stated in the third point BPEL supports orchestration in a manner that combines the activity diagram and the activity hierarchy approaches. An example of this would be similar to the following:

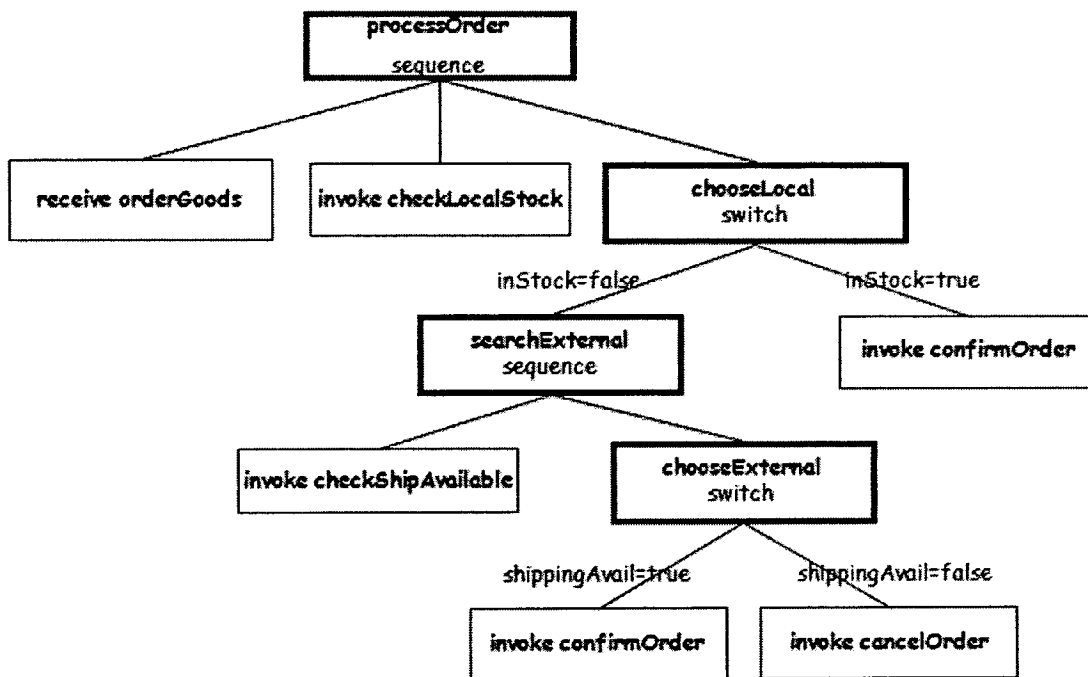


Figure 1.5 Orchestration in BPEL

In use, service composition is equated with orchestration and the tools described for the business process are used internally within a company.

1.2.4 Problem Definition

Orchestration and BPEL help an enterprise manage its business process within a department and among the different departments. Within large organizations there may be hundreds or maybe even thousands of set business processes each serving a purpose. There may be times when a certain target is not satisfied by one singular process or has no specified solution. It would be possible, through composition, to have more than one business process combined in part or in whole to satisfy this target. Parsing the different processes and matching the target can be a cumbersome effort possibly requiring an exponentially large number of matches. The composition problem is therefore an attempt to find a combination of services, an orchestrator, that can produce the behavior required by the target.

1.3 Service Composition

When a target service behavior is not matched by any of the services specified in an organization, the solution is to create a composition from the available services or at the very least check if such a composition is possible.

There are many approaches for the composition problem, ranging from model checking [4], agent planning [5], satisfiability solving [6], and theorem proving [7].

The framework implemented in this paper, first proposed in [9], and usually referred to as the "Roman Model", has been dealt with in many works [10][11][12]. This framework can model two main aspects [10] of service composition: functional requirements and behavioral constraints. There is a third aspect, which is non-functional requirements that can also be incorporated in the framework as explained later. Most solutions, to date, that are based on this framework have either an elevated complexity (e.g. [9]) or used a global approach (e.g. [14]) in which the whole state space, which is exponential in the number of services, needs to be generated beforehand. The fixed point algorithm is such an implementation where the state is generated and then iterated. In each iteration the result is checked for states that do not match the target and they are pruned till a fixed point solution is reached.

On-the-fly algorithm builds a solution incrementally visiting only the relevant state space. While its worst-case complexity is also exponential in the number of services (this is a lower bound, see [15]), in the average case it is much better. On-the-fly improves on previous approaches to the problem and advances the state of the art in service composition by proposing an algorithm that:

- 1) visits states as needed, which allows it to deal efficiently with systems containing a large number of complex services
- 2) is self-contained and can be easily incorporated in any other model.
- 3) can address concerns about quality of service or quality of experience

1.4 Approach and Main Results

The main approach was to compare the performance of the two algorithms under the same conditions. After being implemented and run the computation time and resource consumption could be noted, graphed, and compared. Both algorithms were implemented in Java. The fixed point algorithm has a fixed running time that changes only if the size of the input changes. For a comprehensive result several scenarios were taken into perspective to produce the best and worst cases On-the-fly algorithm might encounter.

For each algorithm and for every scenario a thousand runs were performed and the results were recorded. In every scenario On-the-fly showed significantly faster results without significant overhead. The fixed point algorithm on the other hand was time consuming and resource intensive.

1.5 Thesis Organization

The objective is to build up to the comparison between On-the-fly and fixed point algorithm. Chapter 1 covered the basics of Web Services and the technologies involved. It also broadly defined the web service composition problem, the proposed solutions, and what On-the-fly tries to accomplish.

Chapter 2 formally defines the background for the framework and the solution.

Chapter 3 goes through the code logic and Java implementation.

Chapter 4 wraps up the paper with result interpretation, conclusion, and future work.

Chapter 2: Model and On-the-Fly Algorithm

Before the algorithm is formally set and described we must establish the components of the framework. This section will present the necessary definitions of the components and how they fit together. The line of reasoning followed was studied in [14][17], originally proposed in [9]. The needed components are a target service, a set available services, and an environment. Note that the environment is used in the framework but is omitted from the implementation for simplicity.

Informally we can describe the service composition problem as the following: given a target service and a set of available services, find an orchestrator, if one exists, that delegates requested actions to suitably chosen available services, such that the system will have the same behavior as the target service.

Next we give the formal definition of all components as well as the composition problem.

2.1 Available Services

Environment

First we define the environment:

Definition 1: An environment \mathcal{E} is a tuple $\mathcal{E} = \langle E, \Sigma, e^0, \delta_E \rangle$ where:

- E is a finite set of states.
- e^0 is the initial state.
- Σ is the set of actions that can be performed.
- $\delta_E \subseteq E \times \Sigma \times E$ is the transition relation.

It is convenient to write $(e_1, a, e_2) \in \delta_E$ as $e_1 \xrightarrow{a} e_2$. Such transition means that when the environment is in state e_1 and an action a is performed it will move to a new stat e_2 .

The available services are a set of components that can be *partially* controlled by the orchestrator and interact with the environment.

Each service is defined formally as:

Definition 2: An available service S over an environment \mathcal{E} is a tuple $S = \langle S, \Sigma, s^0, G, \delta \rangle$ where:

- S is a finite set of states.
- Σ is a finite set of actions.
- s^0 is the initial state.
- G is a set of boolean functions that are used to impose constraints on some actions: $g: E \rightarrow \{true, false\}$ where E is the set of environment states.
- $\delta \subset S \times G \times \Sigma \times S$ is the transition relation.

When $(s, g, a, s') \in \delta$ we write $s \xrightarrow{g, a} s'$. A service can make a transition only if the state of the environment allows it. So if the environment is in state e then the service can make the transition only if $g(e) = true$. Given a service $S = \langle S, \Sigma, s^0, G, \delta \rangle$ and environment $\mathcal{E} = \langle E, \Sigma, e^0, \delta_E \rangle$ a trace of S on \mathcal{E} is a, possibly infinite, sequence of the form $(s^0, e^0) \xrightarrow{a^1} (s^1, e^1) \xrightarrow{a^2} \dots$ where $s^i \in S$, $e^i \in E$, $a^i \in \Sigma$, and for all i if $(s^i, e^i) \xrightarrow{a^{i+1}} (s^{i+1}, e^{i+1})$ then $s^i \xrightarrow{g, a^{i+1}} s^{i+1}$ in S with $g(e^i) = true$ for some $g \in G$ and $e^i \xrightarrow{a^{i+1}} e^{i+1}$. Thus a service can make a transition only if the environment can make the same transition, and is in the appropriate state e . A history is a finite prefix of a trace, ending in a state. Given history $h = (s^0, e^0) \xrightarrow{a^1} \dots \xrightarrow{a^i} (s^i, e^i)$ the last state in the history is denoted by $last(h) = (s^i, e^i)$ and the length the history, denoted by $|h| = i$, is the number of actions performed. The reason we say the set of services is *partially* controllable is because of non-determinism. When a service does an "a" transition there could be many possibilities so we are not sure in which state the service will be in after the "a"-transition

Community of Services

A community of services is the set $\mathcal{C} = \{S_1, \dots, S_n, \mathcal{E}\}$ containing all available services together with the environment. The target service to be satisfied as described in the following section will be checked against this community

2.2 Target Service

The target service is the service requested by the client and which the community tries to satisfy by composing a behaviorally equivalent service from the available services. The target service,

denoted by S_t , is defined like any other service over the same environment, except that it is deterministic. As an example we will take the business case mentioned in sections 1.2.1 and 1.2.3 which is called procurement. Suppose a business wants to purchase some goods from a supplier online. Note that when a business buys from a supplier it does so in bulk to acquire special prices and offers. The example to be described will be a watered down version for simplicity.

The first party that wants to purchase will be referred to as the business and the second party offering the items for sale will be referred to as the supplier.

The target to satisfy is the following:

- The business logs into the supplier's online portal.
- The business requests quotes for certain goods.
- The business makes payment for those goods.
- The business logs out of the online portal.

The target would then look like the following:

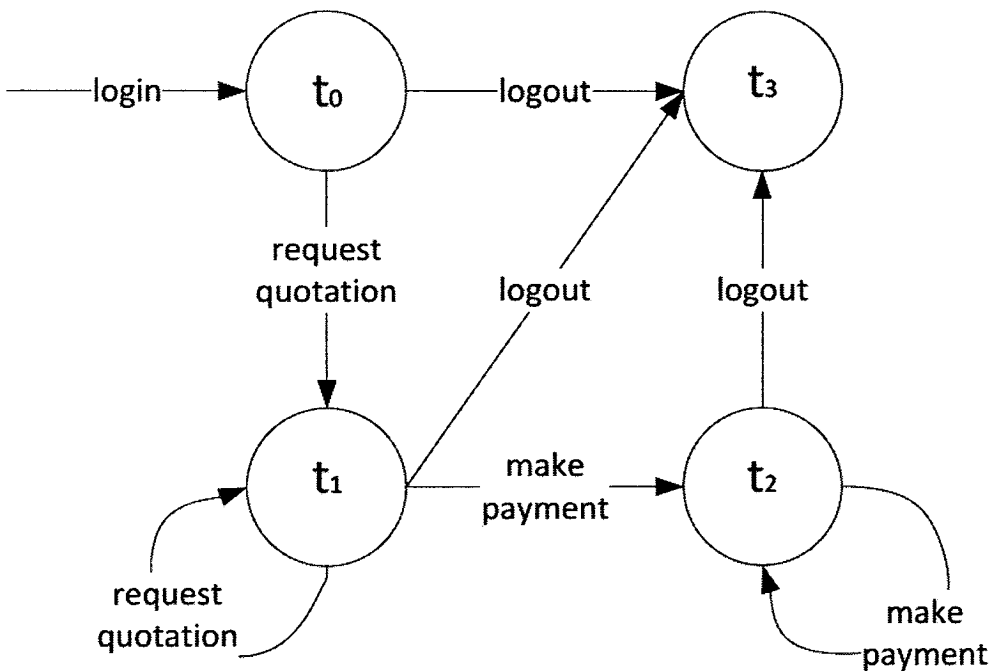


Figure 2.1 Target Service

Naturally the process starts with login. Then the business can request a quotation. At this point which is t_1 the business can request another quotation or make a payment for the one it requested previously. We are assuming that the quotation is similar to a bill in that a business cannot make a payment if no quotation was previously requested. This may not be the case normally but again this was made so for the sake of the example. After at least one quotation request is sent a payment can be made for it. At t_2 the user can make more purchases to buy more of the item quoted or logout if the purchased quantity suffices. This flow can be interrupted by the user at any point by logging out. In case of logging out the transaction is terminated and can only be restarted by logging in, requesting a quotation and so on and so forth.

We assume that such a flow is not readily available. Since it cannot be satisfied by one service an orchestration is required that can replicate this target from the available services which are the following:

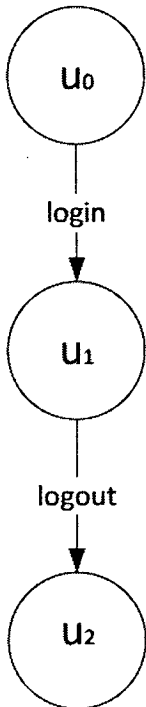


Figure 2.2 User authentication

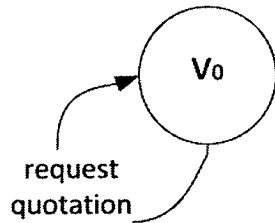


Figure 2.3 Sales

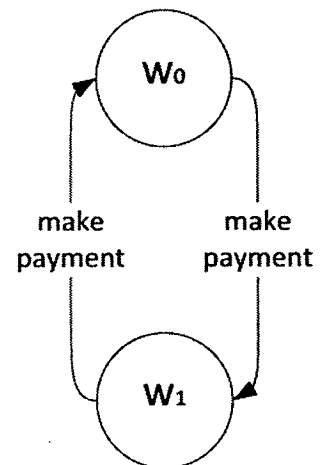


Figure 2.4 Accounting

The supplier is making the services offered in his departments available to his clients to facilitate online transactions. He is using a service to authenticate the users requesting quotations and with this service allowing them to stay logged in and proceed with business as required before logging

out. A separate service is offered by the sales department that can take as many quotation requests as required by the clients. Finally the accounting can take payments for quotations. It is, however, required by the internal business flow of the accounting that these payments be recorded in a certain way to allow for special offers for recurring customers at every other payment.

It is worth mentioning that the environment, although not mentioned above and not taken into consideration in the code implementation, is still effective here. All services have access to the environment where they can store and retrieve data, it is possible for one service to perform authentication, store the results in the environment, and the other services will read the result of the login. The Sales department can then take quotations and store this information for accounting to know the offered amount. Finally the accounting department can take payment for the quotation offered.

2.3 Service Composition

Let $\mathcal{C} = \{S_1, \dots, S_n, \mathcal{E}\}$ be a community of services. A community trace is a sequence of the form $(s_1^0, \dots, s_n^0, e^0) \xrightarrow{k^1 a^1} (s_1^1, \dots, s_n^1, e^1) \xrightarrow{k^2 a^2} \dots$ such that for all $i > 0$ if $(s_1^i, \dots, s_n^i, e^i) \xrightarrow{k^{i+1} a^{i+1}} (s_1^{i+1}, \dots, s_n^{i+1}, e^{i+1})$ then

- $s_{k^{i+1}}^i \xrightarrow{g, a^{i+1}} s_{k^{i+1}}^{i+1}$ and $g(e^i) = \text{true}$ for some g .
- $e^i \xrightarrow{a^{i+1}} e^{i+1}$.
- $s_k^{i+1} = s_k^i$ for all $k \neq k^{i+1}$.

Community histories are finite prefixes of community traces, ending in a state. Given a community history h , we denote by $\text{last}(h)$ the last state in history h . The set of all histories of a community is denoted by \mathcal{H} .

2.3.1 Orchestrator

Given a set of available services the orchestrator is a function $P: \mathcal{H} \times \Sigma \rightarrow \{1, \dots, n, u\}$ that selects a given service $k \in \{1, \dots, n\}$, to delegate action $a \in \Sigma$ to it. The special value u represents the fact that no service can perform the requested action.

Definition 3: [14] The community histories induced by controller P on trace τ is the set

$$\mathcal{H}_{\tau, P} = \bigcup_l \mathcal{H}_{\tau, P}^l \text{ where}$$

- $\mathcal{H}_{\tau, P}^0 = \langle s_1^0, \dots, p_s^0, e^0 \rangle$.
- $\mathcal{H}_{\tau, P}^{i+1}$ is the set of all $i + 1$ length histories of the form $h \xrightarrow{ka^{i+1}} \langle s_1^{i+1}, \dots, s_n^{i+1}, e^{i+1} \rangle$ such that:
 - $h \in \mathcal{H}_{\tau, P}^i$ with $\text{last}(h) = \langle s_1^i, \dots, s_n^i, e^i \rangle$.
 - $P(h, a^{i+1}) = k$. This means that at history h , action a^{i+1} in trace τ is assigned to behavior S_k .
 - $s_k \xrightarrow{g, a^{i+1}} s^k$ and $g(e^i) = \text{true}$ for some g .
 - $s_j^{i+1} = s_j^i$ for $j \neq k$.

Definition 4: We say an orchestrator realizes a target trace τ if for all $h \in \mathcal{H}_{\tau, P}$ if $|h| < |\tau|$ then $P(h, a^{|h|+1}) = k$ and $\text{last}(h) \xrightarrow{ka^{|h|+1}} (s_1', \dots, s_n', e')$ for some k

where the length of an infinite trace is infinite. Basically, an orchestrator realizes a trace if it can always match the target action regardless of the how the system evolved. We say that an orchestrator P realizes a target \mathcal{S}_t if and only if realizes all the traces of \mathcal{S}_t .

In [14] it is shown that a necessary and sufficient condition for the existence of an orchestrator is the existence of a relation between the community and the target service, called an ND-simulation, which we define next.

Definition 5: [14] Let $\mathcal{C} = \langle \mathcal{S}_1, \dots, \mathcal{S}_n, \mathcal{E} \rangle$ be a community of services and $\mathcal{S}_t = \langle \mathcal{S}_t, s_t^0, \Sigma, \delta_t \rangle$ the target service over the same environment. We say that \mathcal{C} and \mathcal{S}_t are *ND-similar* if there exist a relation $Z \subseteq \mathcal{S}_t \times E \times \mathcal{S}_1 \times \dots \times \mathcal{S}_n \times$ such that:

- $(s_t^0, e^0, s_1^0, \dots, s_n^0) \in Z$.
- If $(s_t, s_1, \dots, s_n, e) \in Z$ then for all $a \in \Sigma$ the following holds:
 - if $(s_t, e) \xrightarrow{a} (s_t', e')$ then $\exists k$ such that $s_k \xrightarrow{g, a} s_k'$ with $g(e) = \text{true}$ for some g and $(s_t', e', s_1, \dots, s_k', \dots, s_n) \in Z$.
 - for all $s_k \xrightarrow{g, a} s_k''$ such that $g(e) = \text{true}$ it is the case that $(s_t', e', s_1, \dots, s_k'', \dots, s_n) \in Z$.

Theorem 1: [14] A composition exists if and only if the initial states are ND-similar

Since the union of two ND-simulations is also an ND-simulation then there exists a largest ND-simulation, defined as the union of all ND-simulations. Given a relation $R \subseteq S_t \times E \times S_1 \times \dots \times S_n$ we define a function F over the set of relations over $S_t \times E \times S_1 \times \dots \times S_n$ as follows:

$$F(R) = \{(t, e, p) \mid \forall a, (t, e) \xrightarrow{a} (t', e') \Rightarrow (\exists k, p'. (p, e) \xrightarrow{ka} (p', e') \wedge (t', e', p') \in R \wedge (p \xrightarrow{ka} p'' \Rightarrow (t', e', p'') \in R))\}$$

Where t, e are target and environment states. The state of the n services are represented collectively with p . It is easy to see that a relation R is an ND-simulation iff $R = F(R)$. A typical procedure, similar to the one for classical equivalences and preorders [18], for computing the largest ND-simulation would be to define the set of relations:

$$\begin{aligned} R_0 &= S_t \times E \times S_1 \times \dots \times S_n \\ R_{i+1} &= F(R_i) \end{aligned} \tag{1}$$

Since the transitions systems S under study are finite then there exists a j such that $R_j = F(R_j)$. The largest fixed point, R_j , is the largest ND-simulation one is seeking. Henceforth, this procedure for computing the largest ND-simulation is referred to as "global". The important point to note is that one always starts with R_0 , which is, being the product of all the states, exponential in the number of services. This means one always has to visit all the states in R_0 , the full state space, and more importantly, process all transitions. Clearly this is an expensive

operation and, as will be shown later, unnecessary. In [14] it is shown that after obtaining R_j one can generate all compositions. This is not really necessary since what is usually needed is one composition. It is worth mentioning that other methods solving the composition problem without the use of the concept of ND-simulation also start from the full state space and remove, one by one, non matching states to obtain a solution. In this respect they also are considered "global" algorithms.

2.3.2 Full State Space and ND-Simulation

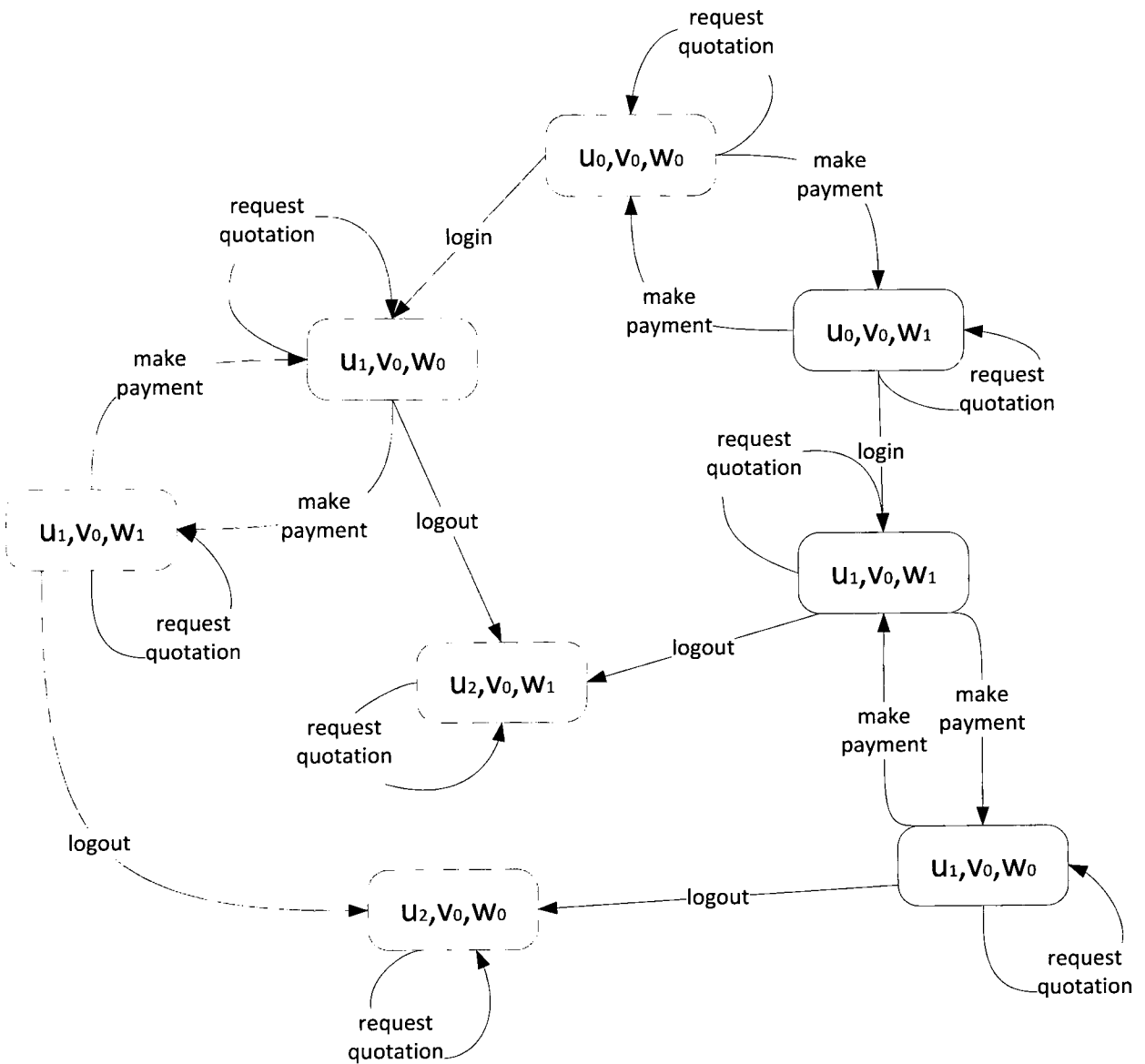


Figure 2.5 State space

The figure above shows the entire state space in all its possibilities. The transitions marked in red are the composition. The existence of a composition indicates that the relation is an ND-simulation. One can see that in this case a fraction of the state space is visited whereas in the global algorithms all the state space, which includes testing all the transitions, will be visited. It is important to note that the environment is not taken into account. With the environment taken into consideration that the make payment transition from (u_1, v_0, w_0) to (u_1, v_0, w_1) and all similar ones can be prevented if at least one request quote transition hasn't been made yet. This would be enforced by the environment where the request quote transition result would be stored.

Furthermore some business rules can be added to the algorithm in finding the ND-simulation to get more preferable business results (better Quality of Service).

2.4 On-the-fly Algorithm

Before delving into the Java code we will examine the code logic and approach of on-the-fly algorithm to find an ND-simulation, if one exists.

Let S_i be the set of states of available service i , E be the set of environment states and S_t the set of states of the target service. The algorithm maintains two relations \mathcal{A} and \mathcal{B} , both initially empty.

The relation $\mathcal{A} \subseteq S_t \times E \times S_1 \times \dots \times S_n$, represents the ND-simulation relation that the algorithm is trying to find between the states of the community and the target. Note that there might be more than one ND-simulation relation.

During the execution of the algorithm states are added and removed from \mathcal{A} . The second relation, $\mathcal{B} \subseteq S_t \times E \times S_1 \times \dots \times S_n$ represents the set of states that were found by the algorithm to not be ND-similar. Because two states found to be not ND-similar cannot become ND-similar at some later stage, states are added to \mathcal{B} but never removed. The set \mathcal{B} is maintained so that a given state is not processed more than once. The algorithm is composed of two mutually recursive functions, NDSIM and MATCH that are described next.

2.4.1 NDSIM

Given a target state (t, e) , and a community state (s_1, \dots, s_n, e) the function

$NDSIM(t, e, s_1, \dots, s_n)$ returns true if and only if the states $\langle t, e, s_1, \dots, s_n \rangle$ are ND-similar.

Basically, $NDSIM$ performs a depth-first search over the state space. When a state is visited for the first time, i.e. not in \mathcal{A} nor in \mathcal{B} , it is assumed to be ND-similar and therefore added to \mathcal{A} .

Then the state is processed by checking that every transition of the target can be matched by a transition of the community.

After a state is processed, if it is found to be not ND-similar, then it is removed from \mathcal{A} and added to \mathcal{B} . Note that $NDSIM$ visits (i.e. adds to \mathcal{A}) states in preorder and processes them in postorder.

Given a target state (t, e) and a community state (s_1, \dots, s_n, e) , the function $NDSIM$ tests whether they are ND-similar. This is the case if and only if for every possible transition of the target state $(t, e) \xrightarrow{\alpha} (t', e')$ the community can match it with an “ α ” transition to a state that is ND-similar to (t', e') .

In other words, given $(t, e) \xrightarrow{\alpha} (t', e')$ the algorithm needs to find for some k , a community transition with the following properties:

1. (P1) There **exists one** s'_k with $s_k \xrightarrow{g, \alpha} s'_k$ with $g(e) = true$ for some $g \in G_k$ such that $(s_1, \dots, s'_k, \dots, s_n, e')$ and (t', e') are ND-similar.
2. (P2) For **every** s''_k , such that $s_k \xrightarrow{g, \alpha} s''_k$, it is the case that $(s_1, \dots, s''_k, \dots, s_n, e')$ and (t', e') are ND-similar.

```

NDSIM( $t, e, s_1, \dots, s_n$ )
if  $\langle t, e, s_1, \dots, s_n \rangle \in \mathcal{B}$  then
|   return false
if  $\langle t, e, s_1, \dots, s_n \rangle \in \mathcal{A}$  then
|   return true
 $\mathcal{A} = \mathcal{A} \cup \langle t, e, s_1, \dots, s_n \rangle$ 
res=true
foreach  $a \in \Sigma$  do
|   foreach  $(t, e) \xrightarrow{a} (t', e')$  do
|       |    $k = \text{MATCH}(s_1, \dots, s_n, e \xrightarrow{a} e', t \xrightarrow{a} t')$ 
|       |   if  $k=0$  then
|       |       |   res=false
|       |       |   Goto Exit
Exit:
if res=false then
|    $\mathcal{B} = \mathcal{B} \cup \langle t, e, s_1, \dots, s_n \rangle$ 
|    $\mathcal{A} = \mathcal{A} - \langle t, e, s_1, \dots, s_n \rangle$ 
|   changed = true
return res

```

2.4.2 MATCH

For every target transition, this function tries to find a community transition that matches it. Given a state $\langle t, e, s_1, \dots, s_n \rangle$, and a target transition $(t, e) \xrightarrow{a} (t', e')$, $MATCH(s_1, \dots, s_n, e \xrightarrow{a} e', t \xrightarrow{a} t')$ tries to find a transition of the community such that property (P1) in section 3.2.1 above is true. It is possible that there could be multiple services that can make an “ α ” transitions from the current state of the community. The $MATCH$ function needs to try them one by one until it finds a match. To this end, $MATCH$ maintains all potentially valid system transitions in a queue.

```
MATCH( $(s_1, \dots, s_n, e \xrightarrow{a} e', t \xrightarrow{a} t')$ )
if  $Q_s$  does not exist then
    create  $Q_s$ 
    for  $i = 1$  to  $n$  do
        if  $s_i \xrightarrow{g, a} s'_i \wedge g_i(e) = true$  then
            ENQUEUE ( $Q_s, i, s'_i$ )
res = false
while  $Q \neq \emptyset \wedge res = false$  do
     $s'_k = DEQUEUE(Q_s)$ 
    res=NDSIM( $t', e', s_1, \dots, s'_k, \dots, s_n$ )
    if res=false then
         $k = 0$ 
return  $k$ 
```


The algorithm maintains a queue that holds all potential transitions of the system from state (s_1, \dots, s_n, e) that can potentially match a target transition $(t, e) \xrightarrow{a} (t', e')$. Since for every system state (s_1, \dots, s_n, e) and target transition $(t, e) \xrightarrow{a} (t', e')$ pair we have a different set of possible transitions the algorithm maintains a different queue for each. Therefore the queue Q_s used in the *MATCH* function, is indexed by s which is a shorthand for $s_1 \dots s_n t e t' e'$. Note that a given Q_s is created when it is needed and keeps the state between different calls of *MATCH*. Once we find that a transition does not match we discard it and dequeue the next possible transition. We keep doing this until a matching transition is found or the queue becomes empty. If the queue becomes empty then there is no match and the function *MATCH* returns the value 0, no matching service is found. In this work we use a FIFO queue but one can as well use a priority queue where the priority is assigned for a given service according to some user preference to implement non-functional requirements, or as a quality of service weight. Also, if the algorithm uses a heuristic based on some already obtained information that makes one transition more likely to succeed, it will be given higher weight. Finally, the queue is defined in such a way that if there are, in a given service k , many s'_k such that $s_k \xrightarrow{a} s'_k$ then *ENQUEUE*(Q_s, k, s') will add the first such s'_k only. This is because from the definition of ND-simulation if $(t, e) \xrightarrow{a} (t', e')$ then it is enough to find a **single** matching transition. The reverse, namely property (P2) is checked in the function *NDSIM*.

The fact that the states are visited in a preorder traversal but processed in a postorder traversal causes a problem, best illustrated with an example, that needs to be handled. Consider the two states $\pi_1 = (t, e, s_1, \dots, s_n)$ and $\pi_2 = (t', e', s'_1, \dots, s'_n)$ with $\pi_1, \pi_2 \in \mathcal{A}$ and $\pi_1 a \rightarrow \pi_2$. This means that π_2 serves as a part of a "proof" that π_1 is ND-similar. Since the algorithm visits the states in preorder and process them in postorder it is possible that π_2 is removed *after* π_1 was processed and used π_2 as a proof. For this reason the algorithm maintains a variable *changed* that is set to true every time a state is found to be not ND-similar. If after the algorithm finishes *changed = true* then there is a possibility that the aforementioned case occurred and the algorithm should be run again:

```
while changed=true do
  | changed=false
  | NDSIM( $t^0, e^0, s_1^0, \dots, s_n^0$ )
```

Note that at the start of every run, or pass, the relation \mathcal{A} and the set of queues is destroyed. The relation \mathcal{B} that keeps track of all non ND-similar states is carried from one pass to the other. As will be shown in the next section it is guaranteed that *changed* will eventually be false and the algorithm will terminate.

2.5 Correctness and Complexity

Let n be the number of available services with each service having N_i states, N_t the number of target service states, and N_e the number of environment states.

Let $N = N_t \times N_e \times N_1 \times \dots \times N_n$.

Theorem: The algorithm *NDSIM* terminates in a finite number of steps and when it does it returns *true* iff $(t^0, e^0, s_1^0, \dots, s_n^0)$ are ND-similar.

Proof:

First we prove the termination. Let $NDSIM_i$ be the i^{th} iteration of $NDSIM(t^0, e^0, s_1^0, \dots, s_n^0)$ and \mathcal{B}_i the set of states that are not ND-similar after $NDSIM_i$ finishes. The variable *changed* is set to *true* iff during the run $\exists (t, e, s_1, \dots, s_n) \notin \mathcal{B}_{i-1}$ and $(t, e, s_1, \dots, s_n) \in \mathcal{B}_i$, meaning that (t, e, s_1, \dots, s_n) was found to be not ND-similar during the execution of $NDSIM_i$. Recall that at no point in the algorithm, states are removed from \mathcal{B} . But if no new state is added to \mathcal{B} then the algorithm stops. This means the set \mathcal{B} is strictly increasing. On the other hand, the total number of states N is finite. Then there is an iteration j such that the variable *change* = *false* and at that point the algorithm terminates.

Next we show that it yields the correct result. Observe that in a given iteration i of the algorithm, we have that if $NDSIM(t, e, s_1, \dots, s_n)$ returns *true* it means that it has finished processing the state $\langle t, e, s_1, \dots, s_n \rangle$ and that $\langle t, e, s_1, \dots, s_n \rangle \in \mathcal{A}$. Also, recall that it returns *true* iff for every a :

1. And for every transition $(t, e) \xrightarrow{a} (t', e')$ there exists a community transition $(s_1, \dots, s_k, \dots, s_n, e) \xrightarrow{a} (s_1, \dots, s'_k, \dots, s_n, e')$ such that $\langle t', e', s_1, \dots, s'_k, \dots, s_n \rangle \in \mathcal{A}$.
2. And for every s''_k such that $(s_1, \dots, s_k, \dots, s_n, e) \xrightarrow{a} (s_1, \dots, s''_k, \dots, s_n, e')$ it is the case that $\langle t, e, s_1, \dots, s''_k, \dots, s_n \rangle \in \mathcal{A}$.

The above two conditions hold in the final iteration, when *changed* = *false* and therefore no $\langle t', e', s_1, \dots, s'_k, \dots, s_n \rangle$ was removed from \mathcal{A} , imply that the relation \mathcal{A} is an ND-simulation.

Theorem: The algorithm *NDSIM* is polynomial in the number of states of a given service and exponential in the number of services.

First recall that $N = N_t \times N_e \times N_1 \times N_2 \times \dots \times N_n$ is the number of possible states of the community and target combined. Since we are doing a worst-case analysis, we assume that all the above states are reachable.

In a single run of *NDSIM*($t^0, e^0, s_1^0, \dots, s_n^0$) each state is considered once. This is because after the first visit it is either in \mathcal{A} or in \mathcal{B} . On any subsequent call it will not be visited again. This means that each iteration of *NDSIM*($t^0, e^0, s_1^0, \dots, s_n^0$) considers at most N states. Next we compute the cost of visiting a single state.

$$\begin{aligned} & \sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |\text{MATCH}(s_1, \dots, s_n, e \xrightarrow{a} e', t \xrightarrow{a} t')| \\ &= \sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}| \end{aligned}$$

The last equality is true because for a given $(t, e) \xrightarrow{a} (t', e')$ the function *MATCH* will process at most $|\{(s_1, \dots, s_n, e) \xrightarrow{a}\}|$ transitions. The above is the contribution of a single state.

Because every state is visited at most once the total cost of one iteration of *NDSIM* is

$$\begin{aligned} &= \sum_e \sum_t \sum_{s_1, \dots, s_n} \sum_a |\{(t, e) \xrightarrow{a}\}| \cdot |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}| \\ &\leq \left(\sum_t \sum_e |\{(t, e) \xrightarrow{a}\}| \right) \cdot \left(\sum_{s_1, \dots, s_n} \sum_a |\{(s_1, \dots, s_n, e) \xrightarrow{a}\}| \right) \\ &= |L_t| \cdot |L_s| \end{aligned} \tag{1}$$

where L_t is the number of transitions of the target system synchronized with the environment and L_s is the number of transitions of the asynchronous product of all the services, synchronized with the environment. To get an idea about the complexity of the algorithm as a function of the

number of services, n , we note that for a given action a , if service i can make $|L_{ai}|$ transitions then the asynchronous product can make $\prod_i |L_{ia}|$. On the other hand, the system cannot make a transition unless the environment does so then we get:

$$|L_s| = \sum_a |L_{ea}| \cdot |L_{1a}| \cdots |L_{na}|$$

In the worst-case every state has an “ α ” transition to every other state. Thus $|L_{ia}| = O(N_i^2)$, where N_i is the number of states in service i . Finally, the complexity of processing a *single* iteration of *NDSIM* is

$$O(N_t^2 \cdot N_e^2 \cdot N_1^2 \cdots N_n^2)$$

Since *NDSIM* is called at most $O(N_t \cdot N_e \cdot N_1 \cdots N_n)$ times on $(t^0, e^0, s_1^0, \dots, s_n^0)$, the total complexity is $O(N_t^3 \cdot N_e^3 \cdot N_1^3 \cdots N_n^3)$. Therefore, the algorithm is polynomial in the number of states of target, environment, or a given services. It is exponential in the number of services. Considering that the problem is EXPTIME-hard [15], this is optimal.

Chapter 3: Implementation

In order to gauge the performance of the algorithm in a real life scenario it had to be implemented and tested in a simulation. Java is fast, highly adopted, and works on anything where a JVM can be installed and so naturally it was one of the top choices for the implementation. Personally I had a background in developing Java Enterprise Applications and could attest to its speed when high performance is required which made it the obvious choice. The implementation was started from scratch and so some basic data structures were required before the algorithm could be implemented. Rather than start from absolute scratch I used some code examples found in [19] and modified them according to what was needed.

3.1 Basic Data Structures

Going from business case to automata requires some transformation. Within a company the available services would be implemented in BPEL. The new target to be matched would also be implemented in BPEL but would have no orchestration. We assume that some transformation is done so that each BPEL is converted into a graph and the operations transformed into edges. After a transformation a real business case could be tested with real life data but for the purposes of our simulation we discuss what the data structures to be transformed into should be.

3.1.1 Action

The Action class represents an edge in a Service with an action label. Each edge consists of two integers which specify from where the edge starts and to where it goes and a String to specify the action label. For our intents and purposes two actions are equal if their labels are equal disregarding the “from” and “to” points. This was implemented in the class by overriding the equals() function and only testing the action label for equality.

3.1.2 Service

The Service class is essentially a directed graph with labeled edges. The vertices are automatically assigned values from zero till “number of vertices” minus 1. Each directed edge is of type Action and is assigned from, to, and label values upon graph creation from the input. Parallel edges and self-loops are permitted. The implementation uses an adjacency-matrix representation, which is an array of Lists of Actions. The integer value of each index represents a vertex and the List at each index contains the neighbors of that vertex. Some variables, to be described in the implementation later, were added for convenience but most notably an adjacency function was added that returns all neighbors of a given vertex for quick iteration. This eases and simplifies the code structure when parsing the adjacency-matrix.

3.1.3 Queue

The Queue class represents a first-in-first-out (FIFO) queue of generic items. Java Generics were used to make the queue versatile for use with other object types later. The implementation supports the usual enqueue and dequeue operations, along with methods for peeking at the first item, testing if the queue is empty, and iterating through the items in FIFO order.

A static nested class labeled Node in the Queue class contains each node’s value and points to the next item. This allows for a linked-list of nodes with easy iteration.

3.1.4 Convenience Functions

Some utility functions and classes were implemented for the test case and are noteworthy. First the services to be tested along with the target are being read from text files (.txt). The text require there to be one service per file with the number of vertices, number of edges, and a listing of from, to, and edge label triplets. A random service generator was also created which effectively generates such text files with random Actions but with the number of vertices and edges taken as arguments. Generating text files in bulk and reading from text files allows test cases to be quickly constructed and documented.

3.2 Implementation

The implementation is a working skeleton of the code proposed. For simplicity the environment was disregarded and a FIFO queue was used. Both can be changed to enforce some business rules or conditions but the general implementation and speed of the algorithm would be unaffected.

3.2.1 Basics

The community to be parsed is passed to the algorithm as a List of Services. An integer array is created to serve as a form of encoding to keep track of the community state. The last index of this array is reserved for the target. As an example let us assume we have two services in the community labeled U and V and a Target. Upon initialization the array would be (0, 0, 0). This indicates that all services have a marker at 0 telling us where each service is being processed. If an action in service U going to node 1 matches an action in the target going to node 2 the state saved in the array would be (1, 0, 2). Relations A and B used to store the solution states and the non-solution states are lists that store arrays of such structure.

3.2.2 Code Breakdown

As stated in the previous section the algorithm mainly operates among two mutually recursive functions. The first one is labeled `ndsim` and its main role is to loop over the target and call `match` to match the target's action. The second is called `match` and it loops over the community and replies to `ndsim` about whether a match was found.

The code important parts of the code will be interpreted line by line before demonstrating an example of how it flows.

Note that the code starts with a function that initializes everything at state 0 and calls `ndsim` expecting a Boolean result.


```

private boolean ndsim(int[] communityState, int targetNode) {
    for (int x = 0; x < B.size(); x++) {
        if (Arrays.equals(B.get(x), communityState)) {
            return false;
        }
    }

    for (int x = 0; x < A.size(); x++) {
        if (Arrays.equals(A.get(x), communityState)) {
            return true;
        }
    }

    A.add(communityState);

    boolean ndsimResult = true;

    for (Action a : target.adj(targetNode)) {
        int k = match(communityState, a);
        if (k == -1) {
            ndsimResult = false;
            break;
        }
    }

    if (!ndsimResult) {
        int[] nonMatch = A.get(A.size() - 1);
        B.add(nonMatch);
        A.remove(nonMatch);
    }

    return ndsimResult;
}

```



Check if state has been visited before and is found in B. Stop check if found because we already found it a non-match



Check if state has been visited before and is found in A. Stop check if found because it was found to be a match



Assume community state is a match



Loop over actions performed by current target node. Send each to match to check if the current community state can match it. If no match found break.



If no match found add community state tot non-match list and Remove it from match list.

```

private int match(int[] communityState, Action targetAction) {
    boolean ndsimResult = false;
    int k = -1;
    BusinessQueue<Service> Q = new BusinessQueue<Service>();

    for (int i = 0; i < community.size(); i++) {
        for (Action a : community.get(i).adj(communityState[i])) {
            if (a.action().equals(targetAction.action())) {
                k = i;
                community.get(i).setActionPointer(a);
                Q.enqueue(community.get(i));
            }
        }
    }

    while (!Q.isEmpty() && !ndsimResult) {
        int[] matchingCommunityState
            = Arrays.copyOf(communityState, communityState.length);
        matchingCommunityState[Q.peek().getIndex()]
            = Q.peek().getActionPointer().to();
        matchingCommunityState[matchingCommunityState.length - 1]
            = targetAction.to();

        Q.dequeue();
        ndsimResult = ndsim(matchingCommunityState, targetAction.to());
        if (!ndsimResult) {
            k = -1;
        }
    }

    return k;
}

```

ndsimResult stores the value from ndsim later on.
 k is the answer returned to ndsim
 Initialize Queue locally for this run
 Loop over community.
 Get the actions of each node in each service.
 If they can match the target enqueue them
 Loop over queue filled from previous for loop. Change the community according to the action specified in the top element
 Dequeue the element as it is no longer needed.
 send the altered community state to ndsim to check for previous match or for actions that will come after the current target action

Putting all possible actions that match the target in the queue also ensures recursion is matched. As an example, if the target action has a node which performs an action “a” and loops back to itself, the community can match it through a cycle of “a” actions stored in the queue.

As a small case study we will take the algorithm through two use cases, one where it will succeed and one where it will fail.

Success case

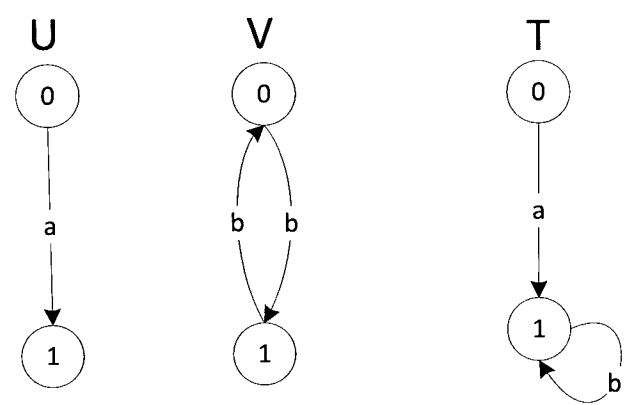


Figure 3.1 Simple Test Case

```

ndsim((0,0,0),0){
  match((0,0,0),0-a->1){
    enqueue 0-a->1 from U
    dequeue top
    ndsim((1,0,1),1){
      match((1,0,1),1-b->1){
        enqueue 0-b->1 from V
        dequeue top
        ndsim((1,1,1),1){
          match((1,1,1),1-b->0){
            enqueue 1-b->0 from V
            dequeue top
            ndsim((1,0,1),1){
              return true//found in A}
            ndsim result true
          }
          return 1}
          match result 1
        }
        return true}
        ndsim result true
      }
      return 1}
      match result 1
    }
    return true}
    ndsim result true
  }
  return 0}
  match result 0
}
return true}

```

Result: A: U0 V0 t0
 U1 V0 t1
 U1 V1 t1

B: Empty

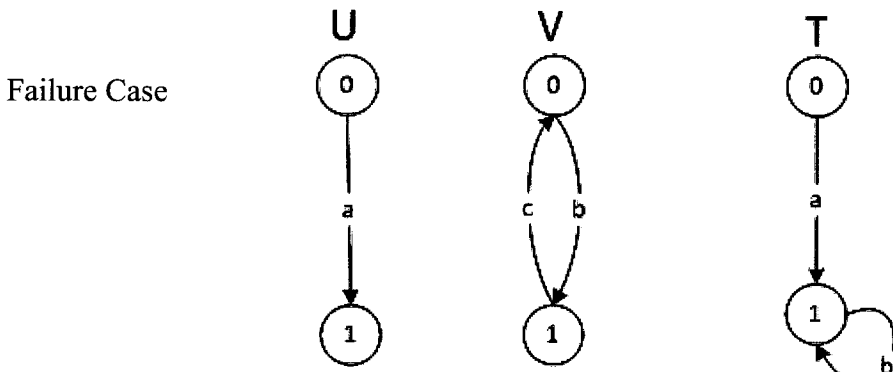


Figure 3.2 Failure Test Case

```

ndsim((0,0,0),0){
  match((0,0,0),0-a->1){
    enqueue 0-a->1 from U
    dequeue top
    ndsim((1,0,1),1){
      match((1,0,1),1-b->1){
        enqueue 0-b->1 from V
        dequeue top
        ndsim((1,1,1),1){
          match((1,1,1),1-b->1){
            return -1}
          match result -1
        }
        return false}
      ndsim result false
    }
    return -1}
    match result -1
  }
  ndsim result false
}
return -1}
match result -1
return false}
  
```

No Result Found!!

3.3 Fixed Point Algorithm

The fixed point algorithm works in a manner similar to the approach described in [14]. First before the actual fixed point algorithm can be run the entire state space must be generated. Then the fixed point algorithm is run on the state space and works according to a set of rules to be discussed next.

The fixed point algorithm runs in several iterations. Every community state will be compared to every target node. The following rules determine if a community state is kept or discarded:

- 1) Every target action must be matched by a community action
- 2) The resulting community and target state after the action must be present in the remaining community from the last iteration

3.3.1 Fixed Point Implementation

The code logic first requires a Cartesian product between all the services and the target.

Taking as an example 3 services U, V, and W, the Cartesian product would be $U \times V \times W \times T$.

Since the number of services is variable a solution is required that runs as long as the number of services. The product would then have to be divided into iterations so that with each iteration one service is multiplied with all the others.

$$U \times V \times W \times T = (U \times (V \times (W \times T)))$$

In every run an outer service is multiplied with the result of those inside.

This generates a Cartesian product of the state space x T.

This result is passed to a function that takes care of the elimination process and recursively calls itself till a fixed point is reached, i.e the result of the last iteration is the same as the current iteration

3.3.2 Fixed Point Code Sample

```
private void cartesianProduct(int[] currentState, int serviceIndex) {
    if (serviceIndex == community.size()) {
        cartesianResult.add(Arrays.copyOf(currentState, currentState.length));
    } else {
        for (int node = 0; node < community.get(serviceIndex).V(); node++) {
            currentState[serviceIndex] = node;
            cartesianProduct(currentState, serviceIndex + 1);
        }
    }
}
```

Each call of function cartesianProduct calls cartesianProduct again recursively and passes to it the next service until the innermost parenthesis, demonstrated above, is reached. Then the function would work its way back through all the calls and register the Cartesian product at every step.

```
void fixedPointCheck(List<int[]> latestResult){
    int solutionSize = latestResult.size();

    List<int[]> newResult = new ArrayList<int[]>();

    for (int x = 0 ; x < latestResult.size() ; x++){
        for (int y = 0 ; y < latestResult.get(x).length-1 ; y++) {
            List<Action> targetActions = target.adj(latestResult.get(x)[latestResult.get(x).length - 1]);
            boolean[] actionMatch = new boolean[targetActions.size()];
            for(int z = 0 ; z < targetActions.size() ; z++){
                for(Action stateAction:community.get(y).adj(latestResult.get(x)[y])){
                    if(stateAction.equals(targetActions.get(z))){
                        int[] newCommunityState = Arrays.copyOf(latestResult.get(x), latestResult.get(x).length);
                        newCommunityState[y] = stateAction.to();
                        newCommunityState[newCommunityState.length - 1] = targetActions.get(z).to();

                        for(int c=0;c<latestResult.size();c++){
                            if(Arrays.equals(newCommunityState, latestResult.get(c))){
                                actionMatch[z] = true;
                            }
                        }
                    }
                }
            }

            if (areAllTrue(actionMatch)) {
                newResult.add(Arrays.copyOf(latestResult.get(x), latestResult.get(x).length));
            }
        }

        if(solutionSize != newResult.size()){
            fixedPointCheck(newResult);
        }
    }
}
```

The second part is slightly more complex and also works recursively.

First the result of the last iteration is taken note of so that after the elimination is over the result is checked for a fixed point.

Then the for loops go through each action of each node in each community state in the entire Cartesian product and try to match that community state to the target. If the target is matched that community state is kept.

For the same input as On-the-fly algorithm :

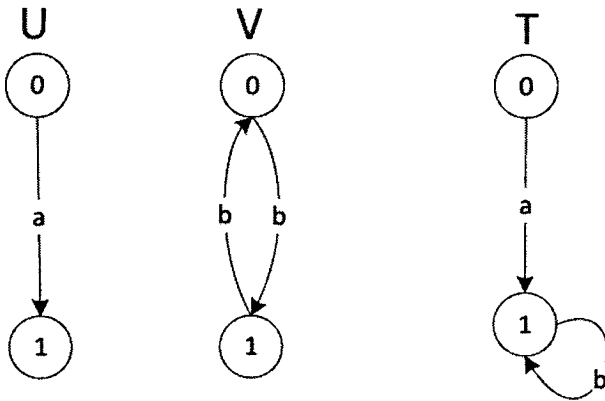


Figure 3.1 Simple Test Case

The fixed point produces the following output:

U0 V0 t0
U0 V0 t1
U0 V1 t0
U0 V1 t1
U1 V0 t1
U1 V1 t1

3.4 Simulation Results and Interpretation

3.4.1 On-the-fly Result

On the fly algorithm gave the following result which is highlighted in the community in red

U0 V0 t0
U1 V0 t1
U1 V1 t1

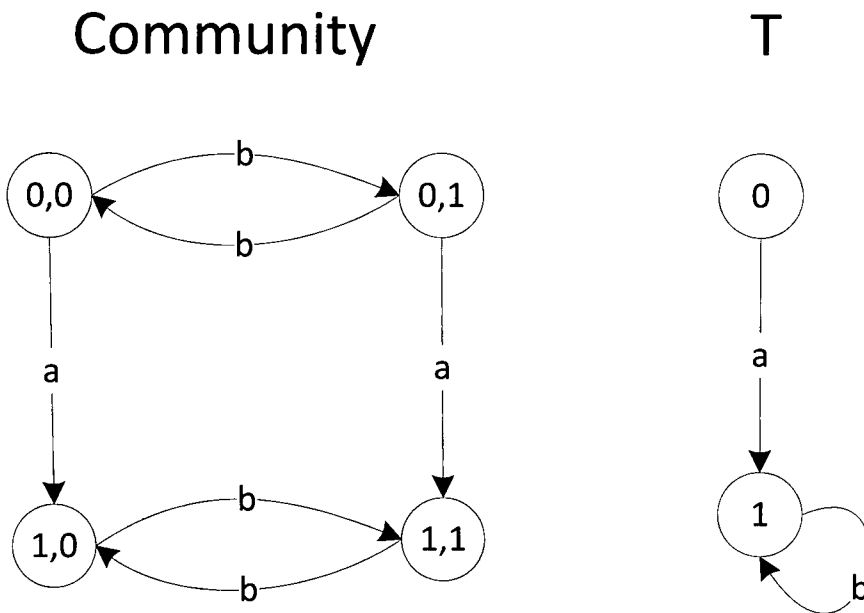


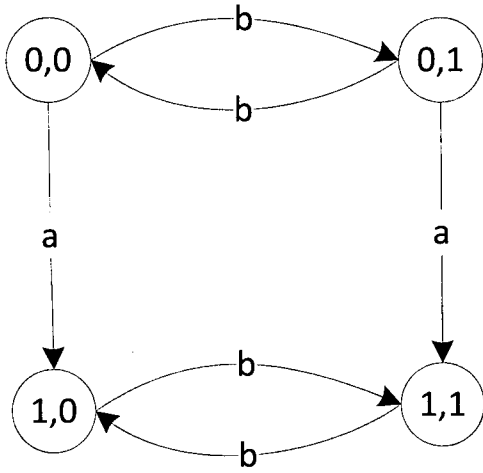
Figure 3.3 On-the-fly result

3.4.2 Fixed Point Result

On the fly algorithm gave the following result which is highlighted in the community in blue

U0 V0 t0
U0 V0 t1
U0 V1 t0
U0 V1 t1
U1 V0 t1
U1 V1 t1

Community



T

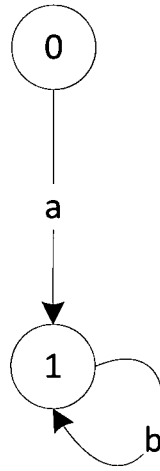


Figure 3.4 Fixed Point Result

On the fly algorithm gives the first result it find whereas the fixed point algorithm gives all community states that can match the target and stay in valid state.

The fixed point algorithm produces the bigger result so intuitively it must take longer to execute.

For more extensive results some cases are taken with speed measurements to see the magnitude of the difference.

3.4.3 Extensive Results

For the first set of results the example used is the one demonstrated in sections 3.3.2, 3.4.2 and shown above in the comparison. The results are based on 1000 runs per algorithm.

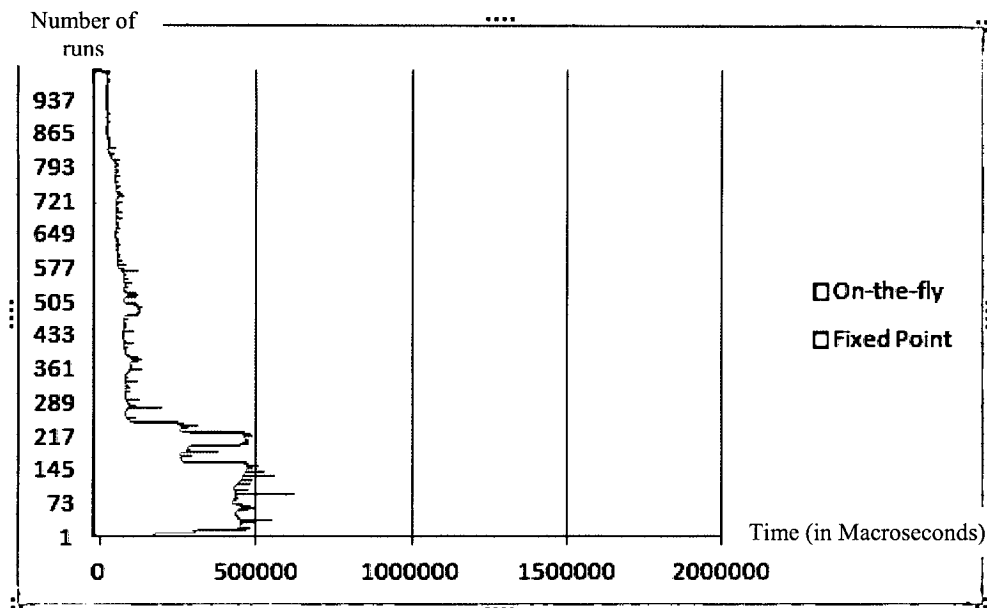


Figure 3.5 Running time comparison for small community

Clearly On-the-fly produced consistently faster results. The x-axis that shows the running time is of the nanoseconds time unit.

One might argue that the difference is ultimately in nanoseconds, the scale of the difference is obviously great in magnitude.

One further thing noticeable is the decrease of the fixed point substantially in running time with each run. This is due to the just-in-time (JIT) compiler that compiles the Java program into machine code after several runs [20].

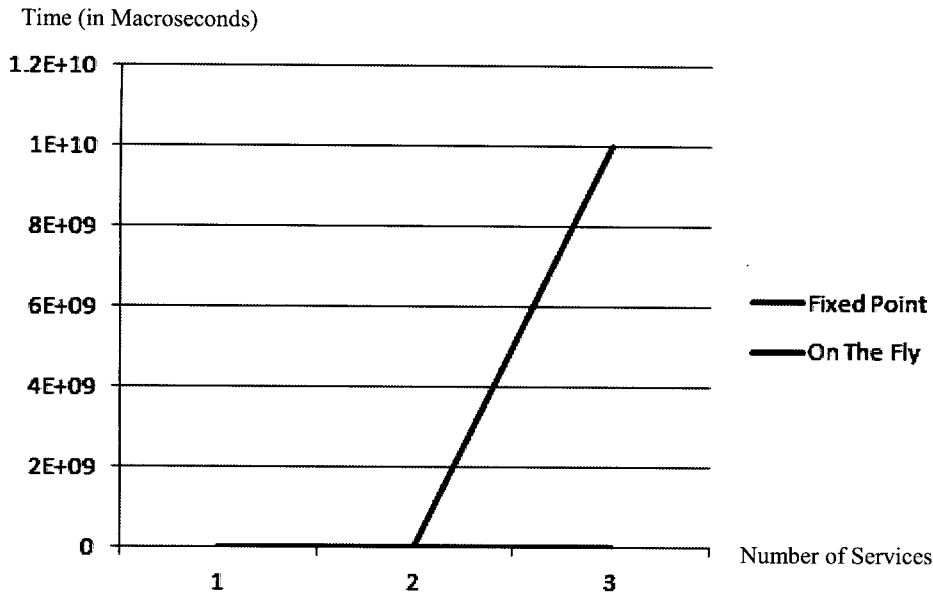


Figure 3.6 Running time vs number of services

Number of Services	Fixed Point	On-the-fly
1	109	25
2	1200000+	96
3	Java heap error	171

For the above graph, services of a fixed size were taken into consideration. The services were created so that there would be the worst case of computation in On-the-fly algorithm.

The services are of size 5 nodes and 4 edges between them.

The Y-axis measurement is computation time in microseconds and the X-axis is the number of services in the community (not counting the target).

On the first run the difference is noticeable although the community is small. On the second run Fixed point took upwards of 20 minutes and did not get result but was stopped because the difference was already large. On the third run Fixed point caused a Java heap error and did not complete.

On-the-Fly consistently got a result and always got it faster. On-the-fly is also less resource intensive and scales better with large problems.

Chapter 4: Conclusions

4.1 Summary of the main results

The two algorithms have been implemented in a similar recursive coding style. In tests On-the-fly produced more consistent results over longer testing and the result was obtained faster. On-the-fly is computationally less demanding which makes it more suitable for more business applications (suitable for applications with a weak client machine) as well as more scalable for larger problems.

On-the-fly can also be combined with some heuristics to get better quality of service. The world is converging toward service-oriented computing paradigms and On-the-fly fits the vision of better performance for faster and more reliable results.

References

- [1] “On-the-Fly Algorithm for the Service Composition Problem”, Farhat H., Feuillade G., 2013
- [2] Gustavo A., Fabio C., Harumi K., Vijay M., “Web Services - Concepts, Architectures and Applications”, Springer Press, 2004
- [3] Giacomo, G.D., Patrizi, F., Sardina, S., “Automatic behavior composition synthesis”, *Artif, Intell*, 196 (2013), 106–142
- [4] Feng Y., Veeramani A., Kanagasabai R., Rho S., “Automatic service composition via model checking. In: Services Computing Conference (APSCC)”, 2011 IEEE Asia-Pacific. (2011) 477–482
- [5] De Giacomo G., Sardina S., “Automatic synthesis of new behaviors from a library of available behaviors”. In: Proceedings of the 20th international joint conference on Artificial intelligence. IJCAI’07, San Francisco, CA, USA, Morgan Kaufmann Publishers Inc. (2007) 1866–1871
- [6] Zahoor E., Perrin O., Godart C., “Web services composition verification using satisfiability solving”. In: Web Services (ICWS), 2012 IEEE 19th International Conference on. (2012) 242–249
- [7] Papapanagiotou P., Fleuriot J., “Formal verification of web services composition using linear logic and the pi-calculus”. In: Web Services (ECOWS), 2011 Ninth IEEE European Conference on. (2011) 31–38
- [8] Rao J., Su X., “A survey of automated web service composition methods”. In: Proceedings of the First international conference on Semantic Web Services and Web Process Composition”. SWSWPC’04, Berlin, Heidelberg, Springer-Verlag (2005) 43–54
- [9] Berardi D., Calvanese D., Giacomo G.D., Lenzerini M., Mecella M., “Automatic composition of e-services that export their behavior”. In: ICSOC. (2003) 43–58
- [10] Balbiani P., Cheikh F., Feuillade G., “Composition of interactive web services based on controller synthesis”. Congress on Services - Part I, 2008. SERVICES ’08. IEEE (July 2008) 521–528