Master Thesis
Service Dependency Analysis based on Process Models and Service Level Agreements

by
Edmundo David Trigos

Dresden University of Technology
Department of Computer Science
Institute for Systems Architecture
Chair of Computer Networks
November 03, 2009

Supervised By:
Prof. Dr. rer. nat. habil. Dr. h. c. Alexander Schill, TU Dresden
M.Sc. Matthias Winkler, SAP Research CEC Dresden
Dr. Ing. Thomas Springer, TU Dresden
Abstract. In the context of Business Processes there is no method currently for modeling inter-service dependencies related to the service’s SLAs. Detection of interservice dependencies at design time is crucial in order to enable effects analysis upon failure or SLA re-negotiation of a service inside a service composition. Existing dependency approaches do not take into account different SLO constraints. The approach is to analyse different approaches for dependency analysis amongst services in a business process. Related work will be analyzed in the area of dependencies and relevant concepts will be related to our approach. Then we will define what dependencies are in the context of a use-case in logistics and define specific dependencies related to this use case. From these definitions we derive specific requirements for the different parts that compose the algorithm. We then develop an algorithm that detects dependencies in the context of their sequence flow and negotiated SLAs. The algorithm is then evaluated with regard to the requirements previously defined.
I confirm that I independently prepared the thesis and that I used only the references and auxiliary means indicated in the thesis.

Dresden, November 03, 2009

---------------------
Edmundo David Trigos
## Contents

1 Introduction ................................................. 1
   1.1 Background ............................................. 1
   1.2 Motivation and Problem ................................. 2
   1.3 Objective ............................................... 3
   1.4 Methodology ........................................... 3
   1.5 Thesis Outline ........................................... 4
   1.6 Acknowledgment ......................................... 4

2 Foundations of SOA and TEXO-IOS .......................... 5
   2.1 Foundations of SOA ....................................... 5
      2.1.1 Basic Technologies ............................... 5
         2.1.1.1 Web Services: WS-Stack, WSDL, SOAP, UDDI . 6
         2.1.2 Service Level Agreement Technologies .......... 7
            2.1.2.1 Background on SLA and QOS ............... 7
            2.1.2.2 Web Service Level Agreement .......... 8
            2.1.2.3 Rule Based Service-Level Agreement .... 8
            2.1.2.4 Semantic Web-Service Level Agreement . 8
            2.1.2.5 WSAG Architecture and Agreement Description 8
         2.1.3 Process Modeling .................................. 11
            2.1.3.1 Business Process Modeling Notation .... 11
            2.1.3.2 Business Process Execution Language ... 14
            2.1.3.3 Differences between BPMN and BPEL ...... 14
   2.2 TEXO and the Internet of Services Vision ............. 15
      2.2.1 Integrating tradable and composable services .... 15
      2.2.2 Service Description under TEXO .................. 16
      2.2.3 Integration of SOA, BPM, and Web Service Agreement 17

3 Dependency Analysis Related Work ........................ 19
   3.1 Background and interdisciplinary nature of dependencies 19
      3.1.1 Applying dependency information .................. 20
         3.1.1.1 Automatic Discovery, Binding and Composition 20
         3.1.1.2 Formal Verification of Processes ........... 21
         3.1.1.3 Root Cause Analysis .......................... 21
      3.1.2 Representing Dependencies ......................... 22
         3.1.2.1 Dependency representation languages ....... 22
         3.1.2.2 Dependency tables ............................. 22
### 3.1.3 Approaches to creating dependency models
- **3.1.3.1 Log analysis approach**
- **3.1.3.2 Data flow analysis**

### 3.2 Approaches to handling service dependencies
- **3.2.1 Control and data dependencies in Business Processes**
- **3.2.2 Monitoring Dependencies for SLAs**
- **3.2.3 Analysis of BPEL data dependencies**
- **3.2.4 Discovering Dependencies between Service Operations**

### 3.3 Conclusion

### 4 Introduction and Requirements for Dependency Discovery
- **4.1 Introduction: What are dependencies**
  - **4.1.1 Example Workflow graph**
  - **4.1.2 Dependency types in Service Compositions**
    - **4.1.2.1 Resource dependencies**
    - **4.1.2.2 Time dependencies**
  - **4.1.3 Introductory Approaches**
    - **4.1.3.1 Introductory Solution**
    - **4.1.3.2 Domain Informed Algorithm Example**
    - **4.1.3.3 Introduction to sub-path algorithm**
- **4.2 Requirements**
  - **4.2.1 Dependency Requirements**
    - **4.2.1.1 Resource Dependency Requirements**
    - **4.2.1.2 Time Dependency Requirements**
  - **4.2.2 Platform Requirements**
- **4.3 Conclusion on Dependency Introduction and Requirements**

### 5 Approach to Dependency Discovery
- **5.1 General Algorithm Approach**
- **5.2 Dependency Detection Algorithm**
  - **5.2.1 Traversal of workflow graph**
    - **5.2.1.1 Depth First Search**
    - **5.2.1.2 AO* algorithm**
    - **5.2.1.3 Mahanti-Sinnakrishnan (M-S) Algorithm**
    - **5.2.1.4 Full sub-path Algorithm**
  - **5.2.2 Dependency Detection**
    - **5.2.2.1 Sub-path traversal**
    - **5.2.2.2 Dependency Detection**
    - **5.2.2.3 Recording of Dependency Model Instance**
- **5.3 Conclusion**

### 6 Implementation and Evaluation
- **6.1 Target Platform**
  - **6.1.0.4 TEXO-IOS and the ISE Development Workbench**
  - **6.1.0.5 Integration into the target platform**
- **6.2 Implementation**
  - **6.2.1 Input Interfaces**
    - **6.2.1.1 BPMN Process Description**
    - **6.2.1.2 SLA Manager**
  - **6.2.2 Output interface**

---

**IV**
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.3 Evaluation</td>
<td>64</td>
</tr>
<tr>
<td>6.3.1 Evaluating Requirements</td>
<td>64</td>
</tr>
<tr>
<td>6.3.1.1 Dependency Function requirements</td>
<td>65</td>
</tr>
<tr>
<td>6.3.1.2 Platform requirements</td>
<td>65</td>
</tr>
<tr>
<td>6.3.2 Performance evaluation</td>
<td>66</td>
</tr>
<tr>
<td>6.3.2.1 Evaluation metrics and processes</td>
<td>67</td>
</tr>
<tr>
<td>6.3.2.2 Evaluation procedure</td>
<td>68</td>
</tr>
<tr>
<td>6.3.2.3 Evaluation results</td>
<td>68</td>
</tr>
<tr>
<td>6.3.3 Limitations</td>
<td>72</td>
</tr>
<tr>
<td>7 Summary and Outlook</td>
<td>77</td>
</tr>
<tr>
<td>7.1 Summary</td>
<td>77</td>
</tr>
<tr>
<td>7.2 Future Work</td>
<td>77</td>
</tr>
<tr>
<td>References</td>
<td>79</td>
</tr>
</tbody>
</table>
6.10 Running times to get different number of sub-paths. ............ 70
6.11 Time to obtain resource dependency pairs ....................... 71
6.12 Time to obtain time dependency pairs ........................... 72
6.13 Time to obtain time dependencies ................................. 73
6.14 Difference between pairs and duplicate pairs in time
               dependencies. .................................................. 74
6.15 Percentage of time in steps of the dependency detection
               algorithm .......................................................... 75
1 Introduction

1.1 Background

Recent advances in enterprise software integration and communication standards amongst IT Systems have made it possible to evolve complex systems capable of inter networking across heterogeneous platforms. These advances also make it possible for enterprises to start bridging the gap between their business processes and the IT Architecture that drives their flow of information.

This evolution is being made possible with IT architecture proposals such as Service Oriented Architecture (SOA). A formal definition is given by the Oasis Group:

“[…] Service-Oriented Architecture (SOA) is a software architecture where functionality is grouped around business processes and packaged as inter operable services. SOA also describes IT infrastructure which allows different applications to exchange data with one another as they participate in business processes. … These services communicate with each other by passing data from one service to another, or by coordinating an activity between two or more services[...].” [Cov00]

A natural extension to this service environment is the proposal of the TEXO Internet of Services (IOS) [H.W], which uses a SOA style architecture and adds facilities for trading services as goods on the Internet. This involves taking into account consumer, provider and Quality of Service (QOS) for the service. According to [H.W] the Internet of Services is:

[…]A trusted Service Ecosystem of service providers, consumers and brokers buying, selling, re-purposing and composing services for different needs ...resulting in a new way of organizing the interaction between partner ecosystem, the community and service consumers […] . [H.W]

In a typical TEXO-IOS scenario, the provider of a service composes and offers services to be used by consumers. This service can have a web-service back-end or be a service offered in the real world. TEXO-IOS also provides service Service Level Agreements (SLAs) which describe the formal agreement about the provided service and its quality of service (QOS) over its functions.

These are some typical steps to provision a service under the TEXO-IOS scenario:
1. The customer selects and configures a service.
2. Customer negotiates and agrees to a quality of service and other service properties such as provider and consumer information.
3. The service is then provisioned to the consumer and the quality of service and other properties are monitored during the service run-time to verify if SLA is violated or not.

The TEXO-IOS goal is to create a service marketplace that leverages these concepts so that providers derive an added business value by composing these services to consumers. Consumers on the other hand, can select these services readily available through the Internet and with quality of service guarantees.

1.2 Motivation and Problem

According to the TEXO IOS vision, service compose many services into business processes under an SLA infrastructure and monitored quality of service so that a client can have a guarantee that the service will be delivered as agreed. Following the SOA architectural style, the units of work performed by these services are divided into activities which are coordinated by the process in order to reach a common business goal [Vas07]. The coordination and collaboration between these service activities inside a workflow sequence implies that activities can depend on others to reach the overall goal of the process: Often these dependencies are implicit in the activity's sequence flow and SLA attributes. Unfortunately any of these services can fail, or their SLAs can change at anytime during the execution of a process. Detection of dependencies between a failing service and the services that depend on the failing service is important because if a service fails or has changed its service guarantees such as its SLA, it can affect other services in the composition. It will also affect the SLAs in the composition, and the composite service itself. Fig.1.1 shows the situation of a failing service with thick dashed arrows from the crossed out service to its dependent services and SLAs.

The provider is motivated to these detect service dependencies to ensure that is service's SLAs are not violated during the process execution, since SLA violations usually mean penalties for the provider and loss of business revenue.

The approach followed is to detect these dependencies during the design of the process, so that the provider will be prepared and know which services depend on a service before they execute. Since the provider knows which services depend on any particular service, if a service fails during process execution, the provider can then adapt the services that depend on this failing service by adapting the depending service's SLA or offering a backup service, which is step four, with the long-dashed arrow pointed at the failed service in Fig.1.1. This last adaptation step beyond the scope of this work. We only deal with the detection of the services that depend on a particular service in the composition.

The problem lies in the difficulty to detect these service dependencies because the dependency information we need to detect dependencies is implicitly coded in the SLA information for each service and the sequence flow information derived from the process description. This implicit information needs
to be integrated and processed and made explicit by an algorithm in order to
detect dependencies. The motivation and problem for this thesis is shown in
Fig. 1.1.

![Fig. 1.1. Motivation and Problem for dependency detection.](image)

To the knowledge of the author there is no other work which we know of
that resolves service dependencies between a set of interdependent services
with regard to specific SLA properties.

### 1.3 Objective

The goal of this thesis is to develop an approach to detect different types of
service dependencies amongst services in a service composition by its their
SLA and sequence flow information inside the process. The result will be a
dependency model that will contain all the dependencies of each service to
other services in the service composition.

### 1.4 Methodology

We will adopt a software engineering iterative approach to construct a mod-
ule that discovers dependencies amongst services in a composition from its
process description as well as its SLA information.

First we investigate the state of the art in the topic area of service depen-
dencies, web service process management and coordination and service level
agreements.

Then a definition of the concept of dependency and derivation of require-
ments for the algorithm is done. Using these requirements, design and imple-
mentation of the algorithm is performed. As output, the algorithm provides
an instance of the dependency model which can be used by other modules to
evaluate the service composition in regards to its dependencies. We will then
evaluate the algorithm and explore its strengths and possible improvements.
The specific methodology follows these steps:

1. Analyze the target architecture based on SOA paradigm and a TEXO-IOS
   platform, and analyze its requirements and constraints.
2. Investigate current literature in the areas of service dependencies, service
   level agreements and business process modeling.
3. Introduce and define dependencies and its types for our case scenario,
   formulate some sample scenarios in the logistics domain, and derive the
   requirements for the algorithm.
4. Design and implement an algorithm that conforms to this set of require-
   ments for detecting different types of dependencies.
5. Integrate the algorithm into the TEXO infrastructure.
6. Evaluate the algorithm regarding performance under various process ex-
   amples.
7. Reach a conclusion and outlook.

1.5 Thesis Outline

The Thesis structure will consist of the following sections:

- Chapter two will discuss the core technologies involved in our work which
  are SOA, WS-Agreement business processes and their extension and inte-
  gration under TEXO.
- Chapter three describes the related work in the area of dependency anal-
  ysis.
- Chapter four introduces dependencies in our problem domain and derives
  the requirements for our solution.
- Chapter five describes in detail the concepts for the algorithm.
- Chapter six provides the implementation and evaluation of the solution
  and integration into the target environment.
- Chapter seven summarizes the work with an outlook for future work in
  this project.

1.6 Acknowledgment

The work for this master thesis was done within the SAP Research CEC Dresden. Thanks to all the wonderful people there, at the TUD and Thanks of course to my family: Veros, Ed, Ly and Cc whose support was invaluable.
Chapter two describes the foundations for the technologies used in our work and the literature related to our solution which will introduce the basic concepts that support our work done in the dependency analysis algorithm.

Here is a brief overview of the chapter:

1. Discussion of SOA concepts and the intercommunication of systems with services as the building blocks is briefly done.
2. Related work done on service-level agreements (SLAs) is discussed.
3. Service composition mechanisms are detailed and related to the SOA paradigm.
4. Previously discussed SOA concepts are mapped to the context of TEXO-IOS.

2.1 Foundations of SOA

New advances in interoperability by using open standards and protocols enable intercommunication between heterogeneous platforms. The widely used service paradigm enables this communication via modular services. The possibilities to integrate functionality from different enterprises in complex ways adds value to the composition of these atomic services.

This interoperability and intercommunication is now being provided by different technical specifications and disciplines such as networks, software development, application integration, and human to machine interfaces. We briefly describe the Service-Oriented Architecture (SOA) architecture style in which our solution is based on, but many other standards to achieve this interoperability exist. SOA will support the description of services, service agreements and the composition of services.

2.1.1 Basic Technologies

In this sub-section, we will look at the SOA principles, and how they enable the intercommunication of resources across enterprises, and how these resources are described as services across the network. Before SOA, much of the integration was done via proprietary components such as DCOM and CORBA, and less scalable rules for data interchange existed, making it hard for enterprises to communicate with each other.
2.1.1.1 Web Services: WS-Stack, WSDL, SOAP, UDDI

The main distinction of SOA and other distributed system frameworks is that it is entity centric, with service entities having many tasks assigned to them. This concept makes for encapsulation of functionality and modularity, which helped to enable this architectural style called SOA. [Vas07].

SOA is also supported by different protocols arranged in layers which sit on top of TCP/IP or some other common inter networking topology. Each layer’s components add some functionality to the layer below, also facilitating a modular architecture. We will explore the relevant layers briefly which together enable the description of services under SOA.

We can see these standard protocols as a stack of layers to enable a framework based on services. The figure below shows the layers that compose the SOA Stack: These services use standard protocols to expose descriptions of their operations and attributes for easier access, querying and inter-collaboration. The description of the major protocols can be described in each layer as follows:

1. UDDI is a standard proposed by OASIS [Cov00] which enables service discovery via public directories and is orthogonal to all layers.
2. The bottom layer describes the transport layer which includes the Hypertext Transfer Protocol (HTTP) is responsible for low-level communication of information across the network.
3. In the layer above, the message representation layer provides a message interchange format that defines message content. A commonly used message format is XML (Extended Markup Language) which is an open standard developed by the W3C and others. In the messaging layer, the Simple Object Access Protocol (SOAP) Protocol W3C recommendation is commonly used to carry the different messages from and to operations, which is the method that SOA encapsulates message exchange.

Fig. 2.1. Soa Stack based on [Vas07]

<table>
<thead>
<tr>
<th>UDDI and Metadata exchange</th>
<th>Service Composition</th>
<th>BPEL</th>
<th>BPMN</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Service Description</td>
<td>WSDL</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Messaging</td>
<td>SOAP</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Representation</td>
<td>XML</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Transport</td>
<td>HTTP, SMTP, TCP/IP, RMI, CORBA</td>
<td></td>
</tr>
</tbody>
</table>
4. The next layer supports the description of web-services with the Web Service Definition Language (WSDL), which is a W3C specification to describe web services as sets of atomic units whose interface can be queried and its functions invoked in a modular way via messages.

5. The top layer describes processes, which integrate the functionality of many services in a collaborating sequence flow with other services to form a service composition (also called business process). This process combines the functionality of many services to reach a common goal.

2.1.2 Service Level Agreement Technologies

Service Level Agreements (SLAs) define a set of mutually agreed obligations between consumer and provider. These obligations are related to the quality of the service which will be provided. SLAs also describe other service properties such as consumer and provider information, guarantees, penalties, functional and non-functional service properties which we will discuss later. Inside an SLA description, the provider of the service guarantees a certain level of quality with regards to specific properties regarding the service. Some technologies to manage service level agreements such as Web Service Level Agreement (WSLA), Rule Based Service Level Agreement (RB-SLA), Web Service-Agreement (WSAG) and Semantic WS-Agreement will be discussed.

2.1.2.1 Background on SLA and QOS

The concepts of SLA and QOS become an important part of an infrastructure that is oriented towards assuring the provisioning of a service with a certain level of quality such as in e-commerce or consumer market services. For example, a provider and consumer can agree on a service to deliver specific goods within a time frame. The provider is bound to provide the goods and fulfill the service properties specified in the SLA, and the consumer is also bound to certain service properties. The SLA is monitored, and if the terms are violated (i.e., goods are incorrect, or time is late) penalties are weighed upon the violating party. Service properties can be differentiated between functional properties, which are the operations available by the service itself and non-functional properties, which are properties related on how well the service performs. One example can be a service operation: "Deliver package. Some non-functional properties of this operation could be the "temperature, or delivery time" of such package. In [O'S06], the author explains that the provision of a service not only considers the service itself, but the limitations, or cost at which this service is being provided. We are interested in how these properties change over time, to see if the agreement has being violated. Usually, if a property is constantly measured throughout the service's lifetime (temperature, time) to monitor its SLA, then it would go under the category of non-functional, since it does not form part of the functionality of the service, but it describes how well the service performs. Both functional and non-functional properties are important parts of an agreement, and we will discuss their importance in later sections. Next, some relevant service agreement standards are discussed.
2.1.2.2 Web Service Level Agreement

The aim of the Web Service Level Agreement (WSLA) project created by IBM is to facilitate the design, creation and monitoring of service agreements between consumers and providers in an automatic way. It has three main goals:

- Define the involved parties as provider, customer, third parties and their functionality exposed to the other parties during the contract period.
- The parameters are specified by metrics describing how an item in the contract is measured, and the responsible party for measuring it.
- Defines each of the parties’ obligations, represented as promises of a party to fulfill an action.

The creation of the WSLA standard has served as basis for other standards that have improved on WSLA shortcomings, with improvements such as providing more information on the consumers and providers, and enabling greater flexibility on the types of data allowed. WS-Agreement is a standard that has evolved from WSLA.

2.1.2.3 Rule Based Service-Level Agreement

The RBSLA is another SLA technology that applies knowledge representation and declarative logic in aims to provide the needed representation, monitoring and enforcement for service contracts, policies and SLAs. The standard interprets and monitors the rules, and changes in contracts can be propagated quickly and robustly to the dependent modules, by updating the rules. Logic constraint rules can be formulated dynamically and the rules can be maintained separately, but a separate interpreter needs to be installed and maintained. [RBS05]

2.1.2.4 Semantic Web-Service Level Agreement

There are also some recent efforts in mapping standards such as the WSLA or WS-Agreement to semantic technologies like OWL such as in Semantic WS-agreement partner selection [OVSH06]. This work promises greater abilities of querying and matching of SLA attributes. Although this solution needs an extra infrastructure to support semantic engines and ontologies in order to successfully express and match agreements in a semantic way, these semantic methodologies hope to soon improve the formal expressiveness and managing of information across many domains, but it is still a maturing area.

2.1.2.5 WSAG Architecture and Agreement Description

The WS-Agreement (WSAG) framework was created by the Global Grid Forum (GGF) [And07] Group, and is aimed at describing an agreement between a consumer and provider of a service. WS-Agreement allows the description of different domains to describe the different parts of an agreement, making it a flexible solution for complex and large service information needed in the TEXO project. The WS-Agreement structure is shown below:

The structure is briefly described here:
The Name will be used to store the SLA and to query it by the SLA monitoring framework, which will be described in the TEXO-IOS section 2.2.

The context includes detailed information about the provider and the consumer.

The terms section includes service description terms and guarantee terms. The service description terms (SDTs) describe static service properties, but whose values are also agreed upon in the SLA and should be monitored. Each service will have its own service description terms, and their values will directly support our work in the dependency analysis.

The guarantee terms section includes service properties which will be monitored and whose range of acceptable values for an agreement is also expressed using constraints. Guarantee terms also include service scopes to which these guarantees hold, conditions under which SLOs will be valid, and business values (i.e., penalties, rewards) related to each SLO. SLOs are described as constraints with their own domain-specific metric.

SLOs are service properties that each party is obligated to fulfill, and the values for different services also constitute the data which will be used to support the dependency analysis.

In Fig 2.3 we can see an example of an SLA created for a truck service with a Truck provider, a transport company consumer, and SLO properties for time and resources:

The WS-agreement is focused on the management of web service agreements, so a series of interactions is outlined between customer and provider,
depicted as a SLA Manager in order to reach a common agreement about the service properties.

A typical of negotiation between consumer and provider is shown in Figure 2.4:

1. Creation of an Agreement Template by the provider.
2. Request of an Agreement Template by the consumer.
3. Negotiation between consumer and provider, where each party reviews, changes and returns the template for the other party to review.
4. When the template is agreed by both consumer and provider, the consumer creates an Offer from the Template.
5. Acceptance and creation of the final Agreement (SLA) from the Offer and placed in the SLA repository for retrieval.
6. Retrieval of the SLA properties from an SLA repository is done for monitoring purposes during service execution.

The WS-Agreement is relevant in our work since the TEXO infrastructure implements the WS-Agreement life cycle between its providers and consumers, and stores the SLA’s properties for later monitoring. The SLA properties of each service will be matched in a programmatic way to derive dependencies between services.

We will analyze exactly how the WSAG architecture supports the management of SLAs under the TEXO infrastructure in the TEXO-IOS section.
2.1.3 Process Modeling

As we have seen in the SOA stack, the top layer in the SOA stack belongs to process modeling, where the integration of service descriptions takes place to create a coherent workflow. The resulting workflow is a business process which will integrate the functionality of all services into a coordinated composite service. For the purposes of our project we will equate workflow, service composition and business process. They all represent the integration of several services into a complex process. The Business Process Execution Language (BPEL) OASIS standard [Cov00] and the Business Process Modeling Notation (BPMN) specification by the Object Management Group (OMG) [OMG05] are notations which describe the composition of services into a sequence flow. We will now describe both notations and their relevance to our work in the TEXO dependency algorithm.

2.1.3.1 Business Process Modeling Notation

The BPMN notation is a graph-based notation oriented towards business analysts which enables the creation of business process diagrams in different
possible levels of detail. This notation can be transformed into an executable
service by a process engine.

The BPMN specification [OMG08] has three main goals:
• Easy creation and understanding of business processes.
• Ability to depict complex or simple scenarios.
• Ability to map to or execute the business process.

Other considerations taken from the paper from Vigneras, et al. [RM06] are:
• The notation is not a structured language, it represents a graphical dia-
gram and its analysis may yield various interpretations. The BPMN de-
sign needs to be carefully structured to avoid ambiguities. BPMN offers
optional data flow support composed of free text with no support for vari-
ables.
• The notation can be directly checked for consistency but conventions need
to be used when creating a diagram since the designer is free to unstruc-
tured flows since it is a diagramming notation. There are no structured
blocks which can be analyzed like in a program. More on the differences
of BPMN and other business process notations and languages will be dis-
cussed in the BPMN /BPEL differences section 2.1.3.3.

In order to familiarize ourselves with the process notations used in our
work, we present a sample process in the logistics area which will be used
and extended throughout the paper:

Business Process design with BPMN

The design of a business process with BPMN offers many constructs to create
a model. There are many considerations that will define a process descrip-
tion: We will now introduce the diagram in the context of the general BPMN
notation and how it will be used to describe a business process:

1. The root element in BPMN, a Pool denotes a process which encapsulates
tasks of entity/organization. LogisticsBusinessProcess in figure 2.5.
2. A Pool has [1..n] lanes which represent a functional area in an organiza-
tion. Our process has one lane Composite Service

Fig. 2.5. BPMN Example [Vas07]
3. A lane has [1..n] tasks. We will mention only relevant tasks in our work.
4. Task execution can be sequential or parallel. Split nodes branch in parallel branches. In our example in figure 2.5, the start event is also a split node begins the process, with \{Truck_DDService, Express_DDService, FoodTransportDDService\} tasks in parallel.
5. Normal nodes just perform a task and continue execution with its successor: \{TGL_HH Storage\}
6. Join nodes synchronize many activities, in 2.5 it is TransGermanylogistics which starts as soon as the three previous processes have ended their execution.
7. Finally, the process splits into three parallel flows: \{HH_Star Truck, HH_LocalSvc, TruckHH_Svc\}. These tasks link to the end task and end of the process.
8. Data objects are optional entities that describe the data flow in a free-text manner. In our example, there are no data flows, since the data will be contained in the SLAs (More information about his later.)

In this way, we can see how the sequence flow describes what activities will execute in what order.

With BPMN, the wide choice of process modeling choices and constructs leaves many options open to the designer of the solution. Please refer to the BPMN specifications for a full description. [OMG08] In this work, only a specific variation of a process description and relevant components of a BPMN workflow are presented. A concrete set of design standards and components used and designed to work with the TEXO-IOS will be outlined in the TEXO-IOS section 2.2.

BPM and SOA

Business Process Management (BPM) is a management discipline which is in part supported by the creation of BPMN diagrams to describe the task to be managed. The area of BPM is vast and includes process design and analysis, integration of manual and electronic services and monitoring tasks. We briefly explain how BPMN and SOA can help to bridge the gap between IT and BPM, and how it affects our decisions when creating BPMN processes under TEXO-IOS.

In an article by Miller [Mil06], he explains that BPM is based on process modeling and performance measurement. The problem is that processes are composed of a collection of resources such as policies, tasks, rules and people that apply their user knowledge, human tasks, and manual workflow checking. This process-centric approach tends to break since there exist many points of failure, making this a problem of integration of these disparate resources.

The SOA paradigm is seen as a solution for this problem, since it specifically maps the tasks inside the processes to specific service end-points. These end-points can then be monitored for any potential failures in the process.

In this way, the role of BPMN interacting with a SOA-based implementation is one of modeling and integrating these disparate tasks which can be executed either manually or as web services. We also see that we need to follow some guidelines to successfully enable a collaboration of SOA and BPM, according to [Mil06]:

13
• Adapt the system to work with the particular implementations of SOA and BPM, since the functionalities may overlap.
• Define coarse grained processes available for refinement by business analysts, so modularity is maintained.
• Consider people as an important part of all processes.

These guidelines will be addressed specially the TEXO-IOS section, but they give us a view on how SOA and human or machine processes can work together under BPM, and BPMN descriptions. We will see that SOA plays a relevant role in monitoring the business process, and our logistics use case involves real services executed in the real world, so these relevant considerations will be discussed in later sections.

2.1.3.2 Business Process Execution Language

In this section BPEL will be shortly described as a candidate to model business processes under the TEXO-IOS. Usually, BPEL is chosen to model processes under these situations:

• BPEL process designers are IT developers, not business analysts.
• The process considers concrete bindings to web-services, and XML data exchanges.
• The BPEL process is created during the design phase of the life cycle, not during analysis. \cite{Jur06}

Since the TEXO-IOS envisions business analysts to design the process, which may be attached to a web service or a real service, we do not include BPEL as part of our solution, which is designed to implement web services only. Only a brief description will be helpful to explain the BPEL business process creation as a whole and some useful concepts used in our work.

BPEL modeling concepts are discussed in order to compare them to the BPMN constructs. BPEL is based on a semi-structured notation (block-based) notation which includes graph and block structures. This means that a series of activities or parallel blocks are encapsulated and analyzed as blocks. This block structure enables formal analysis, interpretation and execution by process engines. BPEL also provides for activity scopes with variables and variable assignment operations. The workflow modeling constructs used in BPEL are similar to the ones used in BPMN. Corresponding constructs for tasks, parallel, conditional execution, events, and fault-handling can all be mapped after careful analysis of both models, such as in: \textit{On the Translation between BPMN and BPEL: Conceptual Mismatch between Process Modeling Languages} by Recker et al. \cite{RM06} Of relevance are the variables which carry implicit data flow to and from each service in the process. Data dependencies can be extracted from this data flow, which is the subject of many works dealing with dependency analysis. The procedural BPEL workflow structure lends itself to data and control flow analysis utilizing various techniques of formal analysis such as will be seen in the related works chapter.

2.1.3.3 Differences between BPMN and BPEL

The difference between BPMN and BPEL have been detailed in works by Vigneras et al. \cite{RM06} and Recker et al. \cite{RM06}:
• The target group for BPEL is at a technical level for developers, while BPMN is used primarily by business analysts.

• The Purpose for creating a BPEL or BPMN diagram also differs in that BPEL is generally to design web-service descriptions to exchange XML information in an executable web-service. BPEL will only accept constructs in an acceptable language syntax, otherwise it will produce a syntax error. BPMN is used to model a process for analysis purposes and uses more intuitive notation, jumps and parallel flows which occur in real world. BPMN will accept diagrams of varying levels of detail and abstraction.

BPMN was chosen as the notation to use in the TEXO platform since our need is to describe the sequence flow of the process that may be backed by a web service or service that executes in the real world. The audience intended to model the TEXO processes is a business analyst, and any specific binding to a web-service would be done in another part of the process design.

In conclusion, business modeling forms an important part of our solution, since it will drive the discovery of dependencies through the sequence flow of the process. Specific details and scenarios of how we implemented our processes will be detailed in the next section and the implementation chapter.

2.2 TEXO and the Internet of Services Vision

2.2.1 Integrating tradable and composable services

This section will provide a basic description of the architecture of a TEXO-IOS platform and how it will relate to our work. In later sections we will extend this description to include details of the technological architecture and implementation of the solution.

TEXO-IOS Architecture

The TEXO-IOS architecture involves the composition, provisioning and monitoring of tradeable and composable services over the Internet. The work done in TEXO spans many areas, including legal, service governance, innovation, usage and engineering. For the area of service engineering which concerns our work, the tasks for modeling and analysis of business processes, is done with the support of three major modules: The ISE Development Environment (ISE Workbench), Service Management Platform (SMP) and the Tradable Service Runtime (TSR).

A general overview of the architecture is given in the diagram below:

Here are some general steps in the design, provisioning and monitoring of a composite service under the TEXO-IOS framework, seen through the context of process modeling, SLA negotiation and execution monitoring:

1. The ISE Development environment enables the progressive definition and refinement of the services to be offered in the marketplace. This is done via model editors and transformations where the service description undergoes adaptations by the different stakeholders in the marketplace.

2. A service composition integrates the atomic services by using a business process modeling tool and offered to the consumer by the provider.
3. The SLA Manager inside the Service Management Platform (SMP) enables the consumer and provider to negotiate and agree to the service level properties of the chosen services. The result is an SLA for each service and stored in the SLA Manager.

4. The Tradable Services Runtime (TSR) will execute and monitor the process through the Process Engine. The run-time service properties will be compared to the properties configured in the SLA, and this way the service monitoring component will detect if a service has violated its SLA.

In this description of a typical TEXO-IOS scenario, we can see how service description, composition and SLA negotiation seen in previous chapters is integrated in order to provide a platform of Tradable services. SOA standards will usually be extended to accommodate business-oriented service infrastructure with QOS and Monitoring of services for TEXO. We now look in more detail into the specific components which are particular to TEXO and how they are related to our work.

### 2.2.2 Service Description under TEXO

Service description is fundamental for TEXO-IOS because services will make up the building blocks to form service compositions. Under TEXO-IOS, The Universal Service Description Language (USDL), specified by Cardoso [Car08].
is used to convey a multi-faceted description of each service which will be used in the service composition. These facets include business, technical as well as operational aspects. Also domain-specific information, functional and also non-functional information about the service provider and consumer, which is a vital part to describe services under a service market place, is part of USDL. Fig. 2.7 below shows the three facets of USDL and examples of non-functional service properties which can be represented for a service.

USDL allows for a description extension to describe domain specific properties of the service. This way we can describe special service characteristics for example in logistics, construction, financial or manufacturing domains [Car08]. Each facet in USDL defines three main facets with many sub-categories and many properties to describe the service unambiguously.

As mentioned in the SLA section, WS-Agreement describes a service in regards to consumer, provider and non-functional properties and relates an SLA descriptor to each service. The direct relevance of USDL in our work is that for each service, its USDL descriptor’s non-functional properties will be mapped to properties inside each service’s SLA from WS-Agreement. Therefore, each service has an SLA with non-functional properties inherited from its USDL description. The SLA Manager will then review these properties with provider and consumer to reach a common agreement, which will form the final SLA for this service, each SLA containing USDL definitions.

2.2.3 Integration of SOA, BPM, and Web Service Agreement

As a summary, we present the different concepts outlined previously and their role in detecting dependencies within a business process with SLAs:

- General SOA Architecture principles support communication for description, integration, composition and access to services.
USDL describes services regarding provided consumer and functional, non-functional and domain-specific properties which will map to SLA information for each service.

WS-Agreement represents information about provider, consumer and service properties through the SLA of each service. Each service in the process will have an SLA associated.

BPMN describes business processes and guides dependency analysis through description of sequence flow of services.

The dependency analysis module will extract each service’s SLA information and match it with the service and process description as the basis for the dependency analysis.

All the components seen in this description of the TEXO infrastructure will support a service marketplace with processes offered and monitored in real-time to the consumer. Our dependency analysis module will in turn provide information of depending and dependent services to support this monitoring and run-time infrastructure in case one of these services fails.
Chapter three describes the work related to dependency analysis in various areas of discipline and how it supports the dependency analysis algorithm under TEXO.

3.1 Background and interdisciplinary nature of dependencies

There is much related work concerning dependency analysis, and a wide range of mechanisms have been proposed to analyze the dependencies of entities in a system. In this section we define dependency analysis in the broad context of coordination theory and how it applies to systems in general, then focus on the specific topic of service dependencies in process descriptions using SLAs.

We will analyze specific research papers in detail in order to compare the different approaches, applications, representations towards dependencies and how they have influenced and supported the design of our model and algorithm.

In the paper “The coordination of dependencies” the authors [MC94] define the coordination of dependencies as a management task. The analogy is made to a well-coordinated basketball team. The team moves transparently in a coordinated fashion, and the players seem to move automatically, but their movements are inter-related. What one member of the team does or does not do affects the team as a whole. This definition of dependency implies certain coordination mechanism of the activities taking place inside a process. In human systems, hierarchical dependencies are found in the way people organize their activities, such as in an enterprise. In computer systems, we also see many examples of dependencies such as in distributed systems, they are in a hierarchy of dependent modules so the system can collaborate properly. What these systems have in common is the need for coordination of components, whenever multiple actors collaborate for a common goal. Coordination processes are very important in our study, since they are the ones that describe how the dependencies are resolved. Different types of dependencies exist, and each type of dependency can be seen resolved by a coordination process. [MC94] Mentions various types of dependencies and their corresponding coordination process. The list is not exhaustive, and we will see that certain
processes and types can be adapted and extended in order to support our solution:

<table>
<thead>
<tr>
<th>Coordination Process</th>
<th>Dependency Type</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Producer-Consumer</td>
<td>Pre-Requisite(Input-Output)</td>
<td>Transport, Tracking</td>
</tr>
<tr>
<td>Simultaneity</td>
<td>Synchronization</td>
<td>Concurrency scheduling</td>
</tr>
</tbody>
</table>

Coordination processes are usually implemented and resolved by a process engine or scheduling process in order to assign the activities a coherent sequence in the flow. In this case, the process engine used in our implementation will execute the activities in the correct order. What is important to understand is that the task of our algorithm will be to detect dependencies given a process description and their SLA information. This means that we have to discover the implicit or explicit coordination mechanisms behind the process based on their control flow and specific domain dependent SLA information to derive dependencies. In particular, we deal with dependencies similar to the Producer-Consumer and Simultaneity dependencies described by Malone et al.\cite{MC94}. More on the specific TEXO dependency discovery algorithms will come in the conceptual section. In conclusion, the types of dependencies outlined in the table will reflected repeatedly in the related work that will be covered next, and in our work. Their applications, representation and approach to represent them will be discussed and briefly related to our work.

3.1.1 Applying dependency information

The authors in these articles have different purposes for the analysis of dependencies, ranging from verification of workflows to enhancement of service discovery and composition, and we highlight the relevant contributions of each related work.

3.1.1.1 Automatic Discovery, Binding and Composition

A large body of work is focused in finding dependencies within a group of web-services so that the process engine can discover, compose and bind them automatically or with more expressive power. Although this application is not directly related to our application, it has provided a lot of research into the analysis of dependencies.

In \cite{VAG04}, the authors propose an approach to generate process compositions semi automatically using enhanced semantic descriptions of Web-services. This enables the description of business processes at higher level abstractions and facilitates the runtime discovery of web-services. In \cite{ZPPN07}, the authors also support service composition by establishing an extended SOA model, where roles are used and are responsible for dependency aware service composition. The relevance in these articles is the explicit creation of an service dependency ontology which describes dependencies between services,
which in our work we also do. Of relevance is also the high-level description for the process which we also implement in order to capture the sequence flow.

3.1.1.2 Formal Verification of Processes

Much work is aimed at formal verification of a service composition. It is known that workflow specifications such as BPMN and BPEL can have structural errors such as deadlocks, synchronization errors, and also control and data flow inconsistencies. Formal verification techniques and their relevance will be discussed in the next section.

The article by [PM07] analyses the workflow using graph search techniques to formally verify a workflow graph. These techniques will be the foundation to our algorithm, so they will be discussed in detail in the concept and implementation chapter.

In [ADW08], the author introduces the concept of verifying the workflow according to compliance rules which are applied to the order and parameters of the activities in the workflow. Relevant is the fact that the sequencing of operations in a workflow can be formally verified, which is an issue that dependency analysis algorithms in general have to consider.

In [QS06], the specification of the workflow is validated using Colored Petri Nets for several correctness criteria such as synchronization and sequencing. The significance of this article points out the fact that workflow representations are often transformed to other representations for verification. In our case, we will use graph analysis techniques to analyze the workflow graph, but an internal mapping still takes place to integrate the SLA information.

3.1.1.3 Root Cause Analysis

In [Bas08], the approach aims at discovering failed services in order to diagnose their impact by looking at the other services it depends on. It identifies dependencies between web-services based on real time log data under the context of the HP SOA Manager system.

In [EK02], the author investigates explicit dependencies amongst components such as databases and web-servers inside distributed environments and it aims to detect the root cause of faulty components with the use of agents that detect component dependencies and a query facility that monitors the dependencies between these components. Of importance is the query facility, which enables querying the distributed system about various aspects of the distributed dependency graph.

In [AM07], the algorithm proposed analyses a workflow graph and has the ability to also query certain parts of the graph to obtain the affected resources if a given resource were to fail. To us is relevant that the analysis methods use workflow graphs like in our solution.

To identify a failure in service, the “MoDe4SLA Approach” in [BWRJ08] identifies relevant dependency relations from a composite service to its components during development phase. Their use of domain-specific Service Level Objectives (SLOs) in a service architecture is very relevant to our work because we also analyze SLAs, SLOs and their dependencies, with the difference that we also calculate the relations between services inside the composition, not just between the global composition and the atomic services.
3.1.2 Representing Dependencies

Dependency information which is gathered from services or components needs to be processed into a format which will enable its use by other components in the system. Processing of this information into a standard format yields a model representation, some of which we examine with regard to our work.

3.1.2.1 Dependency representation languages

Many related works create a new language that more accurately describes dependencies. The approach usually involves the creation of the service representation and the dependency representation it has to other services.

In [ZBH08], the representation of each semantic business service includes the information to link it to its depending and dependent services. The approach also discovers data and control dependencies. More details on the approach will come on the next section since it is of high relevance to our work.

In “Dependency Markup Language” [Tol03], the services and dependencies are declared using a dependency type hierarchy, which describes the aggregation (task-subtask) relationships and generalizations (hierarchies) such as the ones seen in [MC94]. The dependency discovery problem is reduced to navigating the dependency hierarchy which was created, but the drawback is that the dependency ontology and service engine has to be created, which is not a trivial task. DML is relevant to us because it solves the problem of vertical dependencies which will be seen examined in [BWRJ08] and our own work.

In “DAG Synchronization Constraint Language” [QS06], an approach which was inspired by parallel programming, defines a declarative syntax to specify services, their activities and their scheduling in a concurrent environment. Of relevance is the ability to specify synchronization constructs not available in BPEL, such as starting an activity before another ends and the ability to parse the language for correctness. DSCL dependencies such as “Happen Before” or “Happen After” are relevant to our work in modeling explicit dependencies.

The authors of [LA05] propose to match semantic similarities of services, by expressing relations such as Prerequisite, Parallel and Substitute Relationship. These dependencies are described using RDF Triples in XML. The relevance of this article is the use of Pre and post-conditions, which we use in our algorithm as Input and Output, which also is linked to the notion of a service using as input the output of another.

3.1.2.2 Dependency tables

A popular form of representing dependencies is by the use of dependency tables such as in [SY08]. By the construction of a Dependency Table using the metrics from the frequency table, they infer direct and indirect dependencies for their process cases.

In [SJSJ05], a “Dependency Structure Matrix” (DSM) captures accepted dependencies extracted from code by conventional static analysis, and highlights possible deviations in the Software architecture’s dependencies. Specific Algorithms organizes the matrix so it reflects the architecture and highlights...
patterns and problematic dependencies. Of relevance is the table which is used in one of our implementations which stores which dependencies have been matched before in order to save time in the matching process.

### 3.1.3 Approaches to creating dependency models.

The methods used to derive dependency models consider the underlying architecture and components of the solution, coupled with a mechanism to relate behavior, state or sequence relationships between components and represent dependencies between these components.

#### 3.1.3.1 Log analysis approach

In [Bas08], the paper aims to identify dependencies based on real log data of service calls under a SOA service execution engine context (HP-SOA Manager). The dependency discovery module then applies heuristic algorithms in real-time to relate service calls from one component to another and derive dependencies after the system has generated enough calls in the log.

In a similar approach, [SY08] analyses the calls to another component, but this approach is focused specifically in obtaining matching calls between running web-services inside a process. The approach uses an occurrence and adjacency matrices to measure the probability that the existence of an web-service invocation leads to the invocation of another web-service. The approach can adapt to any processes and services since it analyzes calls at runtime. Specifically, both these approaches do not suit us, since in the TEXO context our analysis needs to be done in design-time to be able to prepare a fail-over plan, and needs to unambiguously point to the affected services, not in a heuristic manner. However, the use of the adjacency matrix in [SY08] is useful in order to derive indirect dependencies: “If service A depends on service B and B depends on C, then A depends on C”

#### 3.1.3.2 Data flow analysis

The method of Data flow analysis discussed in [SY08] explains the importance of finding data dependencies, since control and data dependencies are normally ignored by the process environment. The methodology involves the analysis of specific messages among web-services within a composition, specifically BPEL. This method analyses the BPEL process, relates the data messages types between pairs of web-services, and derives a dependency graph. Of relevance is the fact that our work relies in large part in analyzing data dependencies in the SLA service description. Also of significance is the fact that what most work intends to do is to match data types and derive some dependency between their sequence flow and the types of data handled by the dependent services. In our case, we will be handling specific instance data with specific SLA values, not data types, but we also do matching. In [ZZK07], the author maps a BPEL description into a Finite State Machine representation in order to derive data flow dependencies and verify them for correctness. This method considers that variables and data in BPEL links can affect the invocation of sequence flow. Of relevance is the extraction of data...
from the original BPEL representation and creation of an intermediate form. In our work, we will extract data from a workflow and SLA information, analyze the data by using using and intermediate form, and then we generate an instance of our dependency model.

3.2 Approaches to handling service dependencies

In the next sub-sections, we analyze specific related literature in detail which have high relevance in our work.

3.2.1 Control and data dependencies in Business Processes

In [ZBH08], the authors create and optimize control and dependency graphs based on semantic business activities. It also explains in detail how to verify and classify dependency types and process constraints. One problem that the paper cites is that control and data dependencies are ignored by the process environment (in this case BPEL), but they are important to support process modeling, analysis, and execution. Also, different types of dependencies may exist in a BPEL process but they are concealed in its description.

A Business Activity is defined as: Semantic Business Activity = \{ Action, Message, Guard, Input, Output, Precondition, Effect \}

The solution exposed in the paper is that instead of using BPEL alone, Business activities are described with a semantic description that defines a message which is sent across two services, a precondition, effect, input, and output. The activation of the effect depends on the guard condition. With this activity model, a process with control and data dependency graphs is created, and then a minimal model is derived. CVlarification of sequencing constraint and control/data dependencies is also done, in order to analyze a business process correctly.

In relation to our work, the description of inputs and outputs of a business activity is an approach we use to support it inside SLAs for our work. The separation of data and control dependencies is relevant to the treatment of dependencies we need to derive a correct dependency model.

3.2.2 Monitoring Dependencies for SLAs

In [BWRJ08], the authors Bodenstaff et al. propose the modeling of dependencies between a service composition and its component services using domain-specific SLA models. It analyses the internal services that may cause a composite service to fail of why a provider has incurred in SLA violations.

The authors propose a method to:

- Identify dependency relations on the atomic services during design time.
- Derive the impact on the composite service based on step one.
- Monitor these dependencies.
- Structure these results to do impact and root-cause analysis.

As an example a provider obtains data from individual content providers and sends reports to customers. The end provider's overall response time depends on the response time of the service provided by the individual content
providers. In this way, SLA violations in the end-provider and its individual providers result directly in SLA violations in the end provider. The approach derives a dependency graph which specifies on which services the overall SLO depends and how it depends on them.

The individual SLOs are modeled using domain specific models using time and cost dependency models. To model a dependency model based on cost, the function adds the individual cost of each atomic service. On the other hand, to model execution time, the model takes the minimum acceptable execution time, and any value above the limit incurs a violation. As seen in the previous example, the approach needs to encode different logic and semantics for each SLO, but there are common dependency model requirements:

Concerning our work, the use of domain specific models for each metric to describe each dependency calculation, and the use of common requirements to find dependencies for SLOs is highly relevant. This paper is also another example of Petri Net use to examine the high-level process dependencies. In this case, colored Petri-Nets are used. The difference between Mode4SLA and the SLA Manager is that we derive dependencies analyzing the workflow graph directly, and no mapping to Petri Nets is done. Another difference is that we also derive dependencies amongst the services inside the global composition, not only between the global composition and each atomic service. This enables us to find exactly what other services are affected aside from the composition, when a service fails.

3.2.3 Analysis of BPEL data dependencies

In “Analysis of BPEL Data Dependencies” by [ZZK07] the author’s aim is to automate test case generation for composed web-services. Their approach is to derive a Finite State Machine from a BPEL description by mapping the structured activities, and the input and output messages. The mapping is done to insure a correct verification of the process by application of Finite State Machine rules. After mapping BPEL to FSM notation, it analyses the data dependencies in regards to variables, links and the conditional firing of activities depending on variables embedded in the conditions. The paper considers that in BPEL, variables and links can affect the control flow, or variables can be embedded in conditions to control the invocation of links.

It derives a data exchange model, which captures how messages are transferred amongst activities, and the data dependencies are then mapped into a WSA (Web-service Automata) Model. This way, data flow testing can be applied formally to the model. The testing visits each Web Service Automaton activity and resolves its dependencies, detecting any data flow errors.

This paper is an example where the underlying formal model is in the form of a FSA (Finite-State-Automaton), and it is used to formally verify the data flows of variables in the high-level BPEL specification. The paper is relevant because it shows a method of how to map data and sequence flow information into another representation, comparing pairs of services and their data flow. A similar technique will be also used in our approach, since we will map and extract the BPMN and SLA information to an internal intermediate representation and then analyze this information while traversing the workflow graph.
3.2.4 Discovering Dependencies between Service Operations

The paper “An Approach to Discover Dependencies between Service Operations” by [SY08] explains an approach to search and identify relevant service operation dependencies in order to compose services automatically and correctly.

This paper aims to simplify the complex task of searching for relevant service operations in a service composition by means of the discovery of the dependencies between service operations.

The paper declares that in most work in dependencies amongst services most research assumes the existence of manually created operation dependencies. In dynamic service environments such as in the paper, the dependencies are discovered in real time by intercepting service calls. The main contributions of the approach are the retrieval of operation dependencies in occurrences of pairs of service operations called in a certain order.

We can get eight kinds of dependency relationships between two operations, which are the combination of:

- Four kinds of occurrence patterns: Always exist directly, Always exist indirectly, Sometimes exist, Always absent.
- Two kinds of order patterns: Before or After.

The first step is the discovery of operation dependencies by checking whether service operations occur together and how. The derivation of operation dependencies includes the creation of frequency matrices to record how many times an operation has invoked another, and an adjacency matrix, to derive indirect dependencies between a chain of operations. The last step is conflict detection and redundancy checking of the dependency table.

Although the process has to be configured and some heuristics have to be applied in the last phase, it proves to detect most dependencies during the service’s run-time. The relevant areas for us apply to their use of the adjacency matrix which is used to describe if an operation happens before or after another. In our work, we consider checking for duplicate matching calls via a similar adjacency matrix. It is interesting to note that this paper, as in our algorithm, we aim for dependency discovery, but here in the paper [SY08] they do it directly via the analysis of service invocations within services at run-time using heuristics. We do it by analyzing the process graph and SLA information at design time introducing no heuristics.

3.3 Conclusion

In this section, we have described how Dependency analysis is applied in many areas, and each application has its method of representing dependencies in a model, and the approach to represent this model is also varied. Below we describe a diagram showing these facets of dependency analysis:

From the study of the related work in the area of business services, compositions and dependencies, we still have a need for an approach that will detect dependencies amongst services and their SLAs. It is not enough for us to embed dependency annotations into the services similar to the semantic approaches, since the framework does not support semantic annotations. We
Fig. 3.1. Many approaches to dependency analysis

also cannot rely on obtaining the dependency information at run-time from the executing process, since that would be too late to react to any possible failures, and heuristic approaches results stabilize over a period of running time.

The creation of a new language to describe dependencies would also need a process engine to interpret new sequence and dependency constructs, which we do not have. The work done with domain specific functions such as price models and response time seems promising, but we would need to enhance it with dependencies amongst services in the composition, not such between the global service and its atomic services, like it is done now.

In order to bridge this gap between related work and our problem domain, we have taken concepts such as different dependency types, approaches like derivation of service pairs, and representations like inputs and outputs of services in order to design the dependency algorithm for this thesis.
In the next chapter we apply this knowledge in order to define dependencies amongst service compositions using their SLA information. The definitions of dependency, and the requirements which these dependencies bring forth will be detailed in the next chapter.
Chapter four introduces service dependencies in service compositions. Using these definitions, we adopt a systems engineering approach in order to derive the system requirements imposed on the workflow structure, algorithm and target platform.

4.1 Introduction: What are dependencies

In order to introduce the dependency discovery algorithm, we start with a general definition of what a dependency is:

The Princeton University Revised Unabridged Dictionary defines dependency as:

“[...] the state of relying on or being controlled by someone or something else. [...]” [Dic09]

Another definition given is by [Wik09]: A dependency exists

“[...] between two defined elements if a change to the definition of one may result in a change to the other. [...]”

By relating these definitions to our work with business processes and SLAs, we see that dependencies are relations or connections between two entities where both depend on the state of the other. How exactly depends on the context, and our context, deals with the state of the activities inside a business process. We will treat this business process as a workflow graph in order to apply to it workflow graph processing techniques such as traversals.

The first sub-section will introduce dependencies in the area of logistics and two introductory approaches to discover them in the context of an example workflow graph, and we will refer to the same dependencies throughout the thesis.

The example workflow described in this section is an extension of the BPMN process graph seen in the TEXO-IOS section 2.5. Using this example graph we will analyze three introductory examples which will lead us into the formulation of various requirements concerning the final approach. In these examples, for clarity, the services from the first TEXO-IOS business process have been substituted by letters for easier identification.
4.1.1 Example Workflow graph

This example introduces important concepts of how the service composition, the services and related SLA information are related in order to describe the dependencies to resolve our logistics use case. We will examine resources and time properties in the logistics domain which are encoded in each services SLA.

The business goal of the logistics composite service involves a global logistics transport activity. The logistics workflow and its SLA information is configured by following these steps in the TEXO-IOS platform:

1. The provider composes and offers a logistics composite service to a consumer who wants to transport goods from a source location to a destination location by using several atomic transport services to realize this goal.
2. Each of these services picks up specific goods at a specific time and source location and transports them to a specific destination at a delivery time. The information of what goods are transported and at what time makes up each Service’s SLA properties.
3. Provider and consumer will agree on each services SLA properties, and these SLAs are stored for later analysis.
4. An additional activity is considered is a storage point, where goods remain for an agreed period of time before being handed off to the next transport service.

By following the previous steps, an example workflow and related SLAs for each of the services in the workflow can be configured such as in 4.1, and we can start the task of detecting dependencies. Along with the workflow for the business process, table 4.1 shows the SLA information for the services in the workflow.

<table>
<thead>
<tr>
<th>Service Input and Output</th>
<th>Pickup</th>
<th>Delivery</th>
</tr>
</thead>
<tbody>
<tr>
<td>A 1,3 7:00 7:30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B 2,4 7:15 7:45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C 5,6 7:00 8:00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D 1-6 8:15 19:45</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E 1-6 20:00 22:00</td>
<td></td>
<td></td>
</tr>
<tr>
<td>F 1,4 23:01 23:30</td>
<td></td>
<td></td>
</tr>
<tr>
<td>G 2,3 22:30 23:59</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H 5,6 22:15 23:00</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

From Fig. 4.1 description and the SLA information we can infer that the goods to be transported, time and locations are important properties to consider to form the basis for dependency detection.

First, a brief description of the example workflow follows: Each transport service is responsible for picking up resources labeled as input at the time in their service SLA description.
The workflow starts in parallel with services A, B and C, responsible to pickup pallets labeled as input and the time labeled pickup time as output and at the delivery time specified in its SLA.

By analyzing this workflow graph and SLA information of each of its services, two types of dependencies are handled:

1. A resource dependency is a dependency between two services with regard to the common resources they carry. If a service in the workflow does not deliver its agreed products then other services that come after this service that carry the same good will be affected for those resources not delivered.
2. Time dependencies exist between any two adjacent services when a first executing services delivery time comes before the pickup time of its successor’s.

Next we analyze further these dependencies by their classification, description and also references to related work in the area in the study of these dependencies.

4.1.2 Dependency types in Service Compositions

As we have seen, we will deal with resource dependencies that correspond to services which transport the same goods, and time dependencies which deal with the time agreed to pickup and deliver these goods. Malone, et al. ,in
proposes to identify and systematically analyze various dependencies and their related coordination processes:

[...] If coordination is defined as managing dependencies, then further progress should be possible by characterizing different kinds of dependencies and identifying the coordination processes that can be used to manage them [...].

We will classify the dependencies found in the logistics use case. The resource and time properties of the services we analyze expose the following dependency types, which can be related to one of Malone’s coordination processes:

<table>
<thead>
<tr>
<th>Dependency Type</th>
<th>Description</th>
<th>Dependency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource</td>
<td>Products transported by services</td>
<td>Producer - Consumer</td>
</tr>
<tr>
<td>Time</td>
<td>Interval when services execute</td>
<td>Simultaneity / Scheduling</td>
</tr>
</tbody>
</table>

In [MC94] Malone describes transfer dependencies as a type of relationship between activities:

[...] When one activity produces something that is used by another activity, the thing produced must be transferred from the producer activity to the consumer activity. Managing this dependency usually involves physical transportation. [...].

In [LA05] Lin, et al describes the consumer-producer dependency as a relationship based on matching pre-condition and post-conditions of two services. For our purposes, matching the output of a service to the input of its successor service is sufficient to derive a valid resource dependency.

In his work [ZBH08], Zhou et al. describes similar dependencies as dependencies of data required and produced by business activities.

4.1.2.1 Resource dependencies

In table 4.2 resource dependencies illustrated in the example workflow in 4.1 can be classified as a consumer-producer dependency since the output of one service serves as the input to the next.

In [MC94] Malone describes transfer dependencies as a type of relationship between activities:

[...] When one activity produces something that is used by another activity, the thing produced must be transferred from the producer activity to the consumer activity. Managing this dependency usually involves physical transportation. [...].

In [LA05] Lin, et al describes the consumer-producer dependency as a relationship based on matching pre-condition and post-conditions of two services. For our purposes, matching the output of a service to the input of its successor service is sufficient to derive a valid resource dependency.

In his work [ZBH08], Zhou et al. describes similar dependencies as dependencies of data required and produced by business activities.
A business activity B is mandatorily data dependent on another business activity A if the data produced by A is required by B. [...]. [ZBH08]

The semantics of this relationship is very similar to our aim in resource dependencies. Zhou et al., realizes this construct with semantic annotations to his web services, and we do it by examining the SLA information for each service.

Our analysis in related work finds resource dependencies similar to works by Zhou, et al in [ZBH08] and Lin et al, in [LA05] that describe dependencies between two web services by modeling their inputs and outputs. These dependencies are data dependencies which behave similarly to our resource dependencies by matching their types or values across dependent services. Our approach matches literal pallet IDs from one service to another within their SLA properties.

The aim analyzing these resource dependences is to derive an algorithm that will discover all the dependencies which a particular service has on other services with regard to products they are responsible for carrying. Let us examine all dependencies that service F has to other services in Fig 4.1. We refer to the motivation section 1.2:

In case a service fails, we need to know all services which depend on E so we can adapt them. We take service F and detect which services it depends on. E carries resources 1 and 3. If we examine F’s predecessors, we derive the following dependencies for service F:

1. Service E also carries 1 and 3, so we can say that Service F depends on service E for resource 1 and 3, which can be expressed as: \{(F,E): 1,3\}, with the notation: \{(DependingService, DependantService): depending resources \}.
2. Similarly, E’s predecessor, D, also carries 1 and 3, so \{(F,D): 1,3\}.
3. D, which is a join node node, forces us to check all possible resources for a match to the pallets that service F carries:
4. Service A carries 1, so \{(F,A): 1\}.
5. Service B carries 3, so \{(F,B): 4\}.

In conclusion we can say that Service F depends on other services in the workflow using the notation: \{(F,E):1,4\}, \{(F,D):1,4\}, \{(F,A): 1\}, \{(F,B): 4\} Here we have derived service F’s dependencies on other services. To derive the dependencies of other services, we would follow the same approach for the rest of the services in the workflow graph. For the example in Fig. 4.1, Table 4.3 illustrates the rest of the service dependencies regarding resources:

<table>
<thead>
<tr>
<th>Depending Service</th>
<th>Dependencies on service and resource:</th>
</tr>
</thead>
<tbody>
<tr>
<td>A:</td>
<td>{ No dependencies }</td>
</tr>
<tr>
<td>B:</td>
<td>{ No dependencies }</td>
</tr>
<tr>
<td>C:</td>
<td>{ No dependencies }</td>
</tr>
<tr>
<td>D:</td>
<td>{(D,A):1,2}, {(D,B):2,4}, {(D,C):5,6}</td>
</tr>
<tr>
<td>E:</td>
<td>{(E,D):1,2,3,4,5,6} {(E,A):1,3}, {(E,B):2,4}, {(E,C):5,6}</td>
</tr>
<tr>
<td>H:</td>
<td>{(H,E):5,6}, {(H,D):5,6}, {(H,C):5,6}</td>
</tr>
</tbody>
</table>
The information in table 4.3 shows information for all services in the composition and what services they depend on. When a service fails, it is only a question of mapping which services depend on the failing service. Although we have derived this table intuitively, it will help us to compare against the approaches to derive resource dependencies in other sections of this chapter. Taking into account that the services exist in a logistics flow and resources are products to be transported along one of many possible routes, there are many issues and requirements to consider in order to define a valid resource dependency under a logistics flow.

The requirements section will outline these specific requisites for the services which will have such resource dependency, but we end with the formal definition of a resource dependency under the context of our work:

Formally, A service \( S(j) \) depends on service \( S(i) \) for resource \( x \) if \( S(i) \) comes before \( S(j) \) in the workflow sub-path and \( x \) belongs to the set of matching resources between the output of \( S(i) \) and input of \( S(j) \). This relationship can be seen in Fig. 4.2

### 4.1.2.2 Time dependencies

Time dependencies in the context of a logistics scenario deal with a time value in which two services have agreed to deliver or pickup resources. With regards to the coordination processes in table 4.2, they are classified as a simultaneity process, and require the scheduling of the services involved. There may be many types of time dependencies which can be implicit in a workflow, such as parallel or overlapping, happen before, happen after relationships, many of which are described in [All81]. In the context of this thesis, the time relationship that we will discover and is implicit in the sequence workflow is the finish to start relationship.

In a finish to start relationship, a service agrees to deliver a pallet of products at a certain time and the adjacent service cannot start and pickup this same pallet until the first service has finished.

The issue comes when considering the SLA time information of the services, which should also convey this finish to start relationship. Here the SLA information of the service does not guide the discovery of dependencies. The sequence flow defines which service comes before the other, and the only thing we can do with the SLA information is verify that it is correctly specified and matches the sequential flow obtained from the workflow graph. In this thesis, we will detect time dependencies using the sequence information from the workflow graph, and the verification of the SLA information will be done in a separate step.

For example, in the example workflow graph, a time dependence between two services \( D \) and \( E \) exists because they are directly connected and \( E \) does not start until \( D \) ends. It enables the first service to execute fully and fulfill its property \( \text{Delivery Time} \), at which time the Service should have delivered its goods. The next service \( E \) can then fulfill its SLA property \( \text{Pickup Time} \). If that does not happen, the service has incurred a failure.

Formally, A service \( S(j) \) depends on service \( S(i) \) for time if \( S(i) \) comes before \( S(j) \) in the sequence flow and both services are adjacent.

To summarize, we refer to Fig. 4.2 which shows the SLA properties for two adjacent services which exhibit both resource and time dependencies. Service
E depends on D for resources 1 and 2 because D transports pallets 1 and 2 which E needs for fulfillment of its SLA. Service E depends on D with regards to time, since E executes immediately after D. In this case, the pickup time for E comes after the delivery time of D, which follows a correct finish to start relationship between D and E.

Fig. 4.2. Services and their dependencies.

4.1.3 Introductory Approaches

Now that we have introduced what dependencies are, we will illustrate some naive approaches at trying to detect the dependencies previously discussed. Although we have already outlined some dependencies and have solved the resource dependency for the example workflow in 4.1, we still need to derive a robust approach to detecting dependencies in a systematic way. The goal is to derive an algorithm for dependency detection with regards to time and resources. First we will examine these introductory approaches here because they directly apply the knowledge we have gathered about dependencies until now, and will help us formulate some design choices about the dependency discovery algorithm. Regarding the design of the algorithm, the work by Smith in [Sto08] outlines the interaction between design choices and requirements analysis:

[...] In a complex system hierarchy, high level design choices impose requirements on the system sub-components; so something that is a design choice at one level becomes a requirement at the next level down. [...].

The design choices we make with the help of these examples will then be used to derive the requirements in the last section in this chapter in 4.2. These design choices will be described using the introductory approaches.
4.1.3.1 Introductory Solution

To introduce an example of how to discover resource and time dependencies, we briefly discuss a naive solution which involves three steps but does not satisfy our requirements for dependency detection. The steps for the trivial algorithm are these:

1. Take the list of all services in a composition in any order
2. Compare one service against every other. (Taking the cross product)
3. Apply the dependency matching criteria in each comparison, depending on SLO information, and derive dependencies for each pair.

Using this solution, no traversal or process analysis is necessary. The control strategy of this algorithm is just to take all services, compare each service against every other successively, and apply the matching rule or dependency function. The dependency function examines service props in order to decide if there is a dependency between the services being matched.

This trivial approach applies the general steps in the algorithm, although it does not meet the requirements outlined in the previous section.

A drawback with this approach is that it tries to match services which could have no dependencies with each other. Services which have a dependency occur along a path, not across parallel paths or in any given order. In Fig.4.1 we can see that neither F nor G will have resource dependencies between each other, but this algorithm would try to match them nevertheless. In a more complex processes, two parallel paths will never have any resource dependencies in common, since they represent different routes for different resources.

Also, some types of dependencies may warrant the ordering of activities along a path, in which this approach clearly does not distinguish between ordering of activities, and thus may derive incorrect dependencies. A traversal that knows which path it follows helps to assign a relative sequential order to each activity, such as is needed in the dependencies we have seen.

4.1.3.2 Domain Informed Algorithm Example

The idea behind this next approach focuses on the idea that the analysis of the workflow along a traversal path and the dependency detection can take place at the same time. The idea is to traverse the workflow graph, and the information of each activity being traversed guides the algorithm to search for the dependencies. If we run the algorithm specifically regarding the resources associated to each service, the algorithm traverses the workflow path trying to match the output resource in one service to the input resource of the successor service.

In Fig. 4.1 as an example for resource 1, which starts in service A, the algorithm traverses the workflow in order to match resource 1 in each successor service. The path derived is \(\{A-D-E-F\}\), which is the list of all services which depend on resource 1. After analyzing all resources and deriving all sub-paths, via transitive property we can know the services which depend on other services for a particular resource pallet. For example, Service F depends on services E, D, and A for resource 3. F depends on services E, D, and B for resource 1.
This algorithm only creates one graph for each resource, so is efficient, simple to implement and derives direct traversal graphs. Variations of this algorithm could be applied to other dependency types but its particular traversal and matching logic would require different algorithms.

The algorithm only detects dependencies between adjacent services since it only examines the properties of its successor. Further analysis of services would need to be done in order to discover dependencies which are non-adjacent to a service.

The algorithm's main limitation is that it combines the traversal and matching logic, and it can only be used for one type of dependency. Each new dependency type would require a completely different traversal and detection algorithm.

4.1.3.3 Introduction to sub-path algorithm

Some of the lessons learned from trying the previous two approaches is that the analysis of the workflow usually leads to the detection of dependencies along a particular path of the workflow. Resource dependencies are found between pairs directly or indirectly connected services along paths, and time dependencies are also found along a path but of adjacent services. Paths are important because they provide the context of the location of each service relative to the sequence flow of the process. Another important finding from the previous two examples is that service dependencies were never found across parallel paths, so for the dependency types we examine, we can focus on finding dependencies only along particular sub-paths.

Related work in the area of workflow analysis such as [PM07] also analyses specific sub-graphs to be verified in terms of the creation of sub-path instances and then analyzing each instance separate from the flow.

For example, a valid sub-graph which can be derived is: \{A-D-E-F\}, and this sub-path can be analyzed for resource dependencies and time dependencies. This path exposes both time and resource dependencies which are valid with respect to the definitions in 4.1.1. If all sub-paths in the workflow are obtained, then their dependencies can be aggregated and analyzed, and all dependencies for the workflow can then be derived.

These sub-graphs can be analyzed for the relevant dependencies, so we can substitute different dependency algorithms and keep the same traversal approach. This method enables the dependency algorithm to be treated as a modular component which can be called separately from the traversal method. We will detail the approach that obtain all sub-paths and calls the different dependency functions in the next chapter, but these design choices of deriving and traversing sub-paths of a workflow enable the solution to be modular and flexible to changing requirements in the future.

Here we propose the use of an approach to find dependencies by finding all possible sub-path instances and then analyzing using different dependency detection algorithms them in order to detect dependencies. This approach will be discussed in the next chapter as the final solution to detect dependencies, but here we only introduce the approach as an improvement on the introductory approaches and to define the general design of the final solution. This proposal and design choices will shape the following requirements which will formally set the foundation to the approach of dependency detection in the
next chapter. Next we will discuss the requirements of an approach to detect dependencies.

4.2 Requirements

With the introductory definitions of dependencies, general types and some basic examples behind the dependency discovery, we have a basic view on how services can be dependent on each other and how their SLA and sequence information defines these dependencies.

These definitions and examples reveal certain requirements that need to be followed by each dependency detection with regards to the workflow, activities and SLA information in order to derive a valid dependency between services. There are also other requirements in order to integrate our work into the solution platform. These platform requirements will become meaningful in the next chapter, but we present them here because they are also requirements which will affect the overall design choices of the algorithm.

The requirements section addresses these points with regard to detection of dependencies:

1. Requirements that each dependency type dictates, which are derived from the sequence of the activities and SLA properties that are domain specific.
2. The requirements of the platform so that it supports scalable, modular and extensible algorithms which will integrate in a controlled way into the target platform.

4.2.1 Dependency Requirements

Specific dependency functions are important, because they encapsulate different dependencies between services as outlined in table 4.2 in a modular way via an algorithm. In general, each dependency function takes a group of services and their SLAs as parameters in a specific order, compares their relative order and matches certain SLA properties between them in order to see if there is a dependency between them. The order in which these services are matched and which properties are matched is domain-specific and is encapsulated in each dependency detection algorithm. There will be a different dependency detection algorithm for each dependency type. Each algorithm has different requirements depending on the type of dependency is detecting.

4.2.1.1 Resource Dependency Requirements

RDR1: Services follow a transitive and sequential relation.

Dependencies can occur between adjacent or non-adjacent activities. There is a transitive relationship that is defined by the resource that two services have in common. We can say for example that in Fig 4.1 service F depends on E, E depends on D, so that F depends on D for some resources. The dependencies between services follow the direction of the workflow sequentially. In the example, we can say that Service F depends on service A for resource 1, which can be expressed as: {(F,A): 1}. On the other hand, we cannot say the
reverse: Service A depends on service F for resource 1 Because once service A executes correctly, it is not affected by any successor services in the workflow. The traversal algorithm needs to decode the semantics of the relative order of one activity to another and compare them correctly.

**RDR2: Resource dependencies occur between pairs of services.**

The cardinality of describing a resource dependency is one to one such as when F depends on A. This way we can talk about pairs of services which have one dependency in common. Most authors that analyze dependencies between services also use pairs of services for their dependencies. such as in [LA05], [To03] and [ZK07].

**RDR3: The algorithm has to detect and match subsets of matching resources in any activity along the route.**

The algorithm then can determine for each service along the route if there is a dependency to any other service along the route.

**4.2.1.2 Time Dependency Requirements**

Time dependencies define a temporal constraint for two activities in regards to the delivery time of a first activity and pickup time of the next adjacent activity. Fig. 4.3 shows two activities and their dependence with regard to time:

![Fig. 4.3. Time dependency between two services.]

**TDR1: Order of activities is resolved between adjacent activities.**

Time dependencies bind two services via its delivery time and pickup time and the fact that these services are adjacent. One may infer that dependencies are also transitive in nature, meaning they affect its direct and indirect successor(s) but we will only define a time dependency of one service to another only with regard to its immediate successor and not propagated to its other successors because we assume subsequent services can adapt their SLAs to a new time and so do not become dependent on an indirectly adjacent failing service. In this way, only the immediately adjacent service is affected, and the
idea is to adapt the service so that its successor is unaffected. In conclusion, time dependencies require that both services should be adjacent, so the algorithm should be able to detect a service’s successor activity. In Fig. 4.1 we can see that activity $D$ depends on activity $B$ with regards to time, because $D$ is $B$’s successor. In our logistics scenario this means that $D$ will not start executing before $B$ has ended.

**TDR2:** Time dependencies occur between pairs of services.

This requirement is the same as in the resource dependencies, since the dependency detection is made between pairs of adjacent services.

### 4.2.2 Platform Requirements

Another type of requirement considers the design and implementation of the algorithm so that it integrates into the target platform in an extensible and modular way. We will now describe the different platform requirements which will be proven in the evaluation section: Because the algorithm interacts with other modules and resides in an integrated environment, we consider the following relevant requirements so it integrates in the target platform. Other requirements commonly considered in systems engineering projects are security and reliability which are provided by the host platform and are not relevant in our case.\cite{Jal08}:

1. **PR1:** The algorithm should be integrated into the platform.

   This involves the interaction with other plug-ins in an extensible and modular way to realize our goal. The platform is a TEXO ISE Workbench plug-in Environment composed of collaborating plug-ins. We can refer to the TEXO-IOS section for a general description of the platform\cite{Jal08}.

2. **PR2:** The algorithm performs its different tasks via interchangeable modules.

   From the process example and descriptions earlier, we can see that there are two distinct parts to the algorithm, which is the derivation of subgraphs. Another part is the dependency analysis, that uses the traversal in order to derive a list of pairs from each sub-graph and apply the dependency function on each pair. Both major parts of the algorithm will be discussed in the next concepts chapter. Modularity will enable us to test, interchange these functions independently of one another without the need to change the other parts.

3. **PR3:** Extensibility enables the algorithm to grow its functionality.

   This includes accepting different types of input formats. The algorithm should recognize different types of process and SLA descriptions. It should also be extensible with regards to being able to extend the dependency algorithms with new functionality.
4. **PR4: The algorithm should support processes with an increasing number of activities.**

Scalability refers to the concept of the algorithm to grow its capacity when many clients or processes are involved. In our case, no batch processes or multiple clients or processes are expected, but it is required that the algorithm support processes of growing sizes and process them in an acceptable amount of time.

### 4.3 Conclusion on Dependency Introduction and Requirements.

In this chapter we have extended our logistics use case scenario and the description of its dependencies. Characteristics of each dependency type, its activities and the workflow in general have been detailed and some introductory approaches have been illustrated in order to show the idea and logic behind the dependency detection algorithms.

Improving on these introductory approaches we have introduced an approach based on deriving sub-paths and applying separate dependency detection algorithms. By using this approach we derived the requirements which such an approach needs in order to detect dependencies in the context of this thesis.

What is needed now is a methodology to integrate the traversal of a workflow while applying the specific dependency criteria we have explained in this chapter in order to detect dependencies. The specific concepts and methodology on how to traverse the workflow graph and how to match specific SLA information between services in the workflow graph will be explained in more detail in the next chapter.
Chapter five discusses our approach to dependency detection, and is divided in two sections. The first section introduces a general approach to algorithm design which will be applied to ours. The second section proposes the dependency detection algorithm which is the core contribution in our work. The algorithm is divided into two major parts: The first part discusses the workflow analysis and sub-graph derivation. The second sub-section deals with sub-graph traversal and dependency functions.

5.1 General Algorithm Approach

In the previous chapter we have detailed the requirements that the dependency detection algorithm should fulfill. There are some assumptions and constraints which we take into account that will form part of the solution, which we will first discuss. Then follow a general approach to design an algorithm that will discover dependencies in such a workflow. In [Nil84], Nilsson presents a general algorithm for problem solving in data sets involving the iterative application of rules to reach a common goal:

```
1: Let Data = Problem domain variables.
2: Loop Until: Data reaches a termination condition
3: BEGIN
4: Select some rule r from a set of rules ‘R’
5: Result = Apply r to Data
6: OutputData = Result
7: END
```

Fig. 5.1. Pseudo-code for general algorithm.

Nilsson’s algorithm follows an approach to divide the problem domain into different steps, where each step derives an intermediate result in each iteration. In our algorithm, we have two major processing steps, the first step derives a set of sub-graphs from the input process description, and the second step analyses the results given in the first step of the algorithm.
Nilsson introduces the concept of control strategy in which specific rule sequences are applied to the data set, and the results given depend on which rule is applied at some step of the algorithm. This concept is shown in line four and five in [5.1] where the decision on which rule to apply can take into account results from past iterations. The result in line five may be used to guide rule applications in later iterations.

The control strategy is important when we analyze graph workflow graphs such as in service compositions, since analysis methods usually apply a sequence of rules to the data in one step, and the next step uses these results which serve to guide the next iteration in the traversal of a workflow graph.

A control strategy can be uninformed, which means the algorithm applies rules and takes certain steps without regard to the data set, or informed, which acts depending on the intermediate feedback it sees from the application of these rules. We will see that some steps in the algorithms we create are uninformed, but some are informed.

Finally, the end of the algorithm is signaled in line two of the general algorithm [5.1] once a satisfactory termination condition is reached. This condition can be triggered by the rules or the state of the data set during the end iteration. This general algorithm approach will serve to outline the approach of our algorithm.

In the last chapter in 4.1.3, two naive approaches to discovering dependencies under the logistics and SLA context were illustrated and the final full-sub-graph algorithm was only introduced. This served to outline the approach briefly, set the foundations to derive the requirements and expose the shortcomings of the introductory approaches, which are solved in the final algorithm which we describe next.

### 5.2 Dependency Detection Algorithm.

So far the introductory approaches to detect dependencies in 4.1.3 fulfill some requirements of the solution, but have some shortcomings in efficiency, modularity and can derive false results. Here we show our algorithm concept of successive application of rules, and taking into account design choices and requirements derived in the requirements section in 4.2, we now improve on these introductory examples and propose the dependency detection algorithm. Following Nilsson's structure of problem solving of a data set in Fig. 5.1, the specific pseudo-code to analyze dependencies is shown in Fig. 5.2.

Just as Nilsson's problem solving approach includes a satisfactory termination condition, the termination condition for our algorithm is: "All services have been matched using the appropriate dependency function along a path and the relevant dependencies have been derived". The dependency detection algorithm is divided into two major steps: The first step given in 5.2 deals with the traversal of the workflow graph to derive all possible sub-graphs. Different approaches to graph traversal by other authors will be compared and contrasted to our approach. The second major step of the algorithm analyses each of these sub-paths, performing dependency analysis by matching of SLA information of relevant activities in order to derive a dependency.

Some assumptions should be made clear before we describe the algorithm in detail.
Fig. 5.2. Pseudo-code to detect dependencies

Here are the detailed steps of the algorithm which we describe in the next sub-sections:

1. Traverse workflow graph to derive all sub-graphs.
2. For each sub-graph, analyze dependencies.
   a) Derive relevant sets of services within the current sub-path.
   b) Call the relevant dependency function with each set of these services.
   c) Create entry in the dependency model if a dependency exists.

Fig. 5.3 illustrates the steps in the process of detecting dependencies:

Fig. 5.3. Dependency Detection Algorithm.
5.2.1 Traversal of workflow graph

The aim of the first step in the dependency algorithm is to traverse all services in the graph, and obtain all possible linear sub-paths from the workflow. The idea behind this approach follows other business process analysis methods such as [PM07], [AM07] and [ADW08] and, [Nil84] that also extract sub-paths from the complete workflow.

For this step, we have analyzed several approaches to traverse a workflow graph such as Depth First Search, AO* [Nil84] and MS [PM07] algorithms to explore the entire workflow graph and extract all possible sub-graphs. We describe the results below:

5.2.1.1 Depth First Search

The Depth First Search algorithm provides an effective mechanism to traverse all nodes in a graph and obtaining a linear path or path segment by traversing the graph from the starting node and then expanding the nodes of the graph until it finds an end node. The algorithm examines all nodes in the graph thanks to its backtracking mechanism.

Cormen provides an explanation of the algorithm in [CLRS01]:

1. The traversal begins with the start node of the graph and traverses forward to the node's successor.
2. Whenever it finds a split-node, it remembers it for later traversal. It goes deeper until it finds an end node.
3. When the algorithm finds an end node, the algorithm backtracks, going back to the last partially explored deepest (PED) split node, which has been remembered in the last step, and chooses the next unexplored child from this node.
4. Keep traversing like in step one starting with this PED-child node.
5. Loops successively until it finds no more children in any of the PED nodes visited.

Using Fig. 4.1 as an example graph, the traversal will proceed as follows:

1. First traverse through the start node’s successors until it reaches node F, remember the split nodes encountered and their children for later traversal. This creates sub-graph { A-D-E-F }.
2. Since F is an end node, the algorithm backtracks to the current PED split node which is E and chooses its next child, which is G. Similarly after traversing G, goes back again to E, then chooses H.
3. The next PED node is the start node, and its next child is B, producing {B}
4. The algorithm consecutively iterates to produce each sub-path instance in the table in 5.1

Each iteration of the algorithm produces a sub-path instance. The list of sub-path instances produced is shown below.

In regards to finding full paths, it will sometimes obtain segments of unexplored children, which are not full-sub-paths, (such as in the partial and repeated segments G and H) For the reason that the sub-paths derived may not be full paths, we cannot use this algorithm to match services along them.
As seen in table 5.1, nodes B, C, G and H always are alone because there is no method to link them to their corresponding path. In order to link these nodes to the sub-path to where they belong, a method to obtain full sub-paths needs to be derived.

5.2.1.2 AO* algorithm

The AO* algorithm by Nilsson in [Nil84] is used in the context of AI systems and is geared to finding the sub-path instance with the minimum cost for a given AND/OR path.

It uses the concept of expanding nodes in the graph iteratively and evaluating the cost of its successors. It marks the successor with the lowest cost. These markings will then be followed in the next step in order to trace down to the current lowest cost path, and then continue expanding the node in the partial lowest cost graph. The algorithm ends when the graph hits a terminal node. A terminal node is one that has no outgoing connectors.

This algorithm is not relevant to us in other respects, since the author’s intent is to iteratively search the graph and calculate only one lowest cost path, while avoiding all the rest. For our purposes, we need to exhaustively match every service with with other services along the same path and all possible paths. This behavior is afforded by the next algorithm: (M-S).

5.2.1.3 Mahanti-Sinnakrishnan (M-S) Algorithm

The Mahanti-Sinnakrishnan (M-S) Algorithm described in “Applying Graph Search Techniques for Workflow Verification” [PM07] belongs to a class of algorithms used to verify the structure of a graph by searching for deadlock and synchronization problems in the workflow. It provides a mechanism to verify AND/OR graphs such as encountered in business processes.

The approach analyses the workflow graph and in each iteration creates a single instance sub-path in the following steps:

1. In each iteration, Create an Instance Sub-path. (CIS)
   a) A sub-path instance is created starting from the current PED node child given from the previous iteration or start node.
   b) Expand and mark only the first OR node child, and all AND nodes along the workflow until it hits an end node.
   c) If the current node has already been expanded, do not create this sub-path instance.
2. Then we Verify Instance Sub-path(VIS)
a) Follow the marks created in the first step that represent an instance sub-path except when they have been traversed before.
b) Check for deadlock or synchronization condition. If error, abort, else continue.
c) Move the mark to the next child of the PED-node for the next iteration.

Here is a sample traversal using MS algorithm using 5.4 as an example. The diamond shape in the edge means that this edge has been expanded.

1. In 1, CIS traverse A, since it is a split node, go through the first child and mark it, go through B, (normal node) so just traverse it. D is a split node so it marks the first child, goes through E and then G is the end. VIS follows these markings to guide itself to create the sub-path: A-B-D-E-G, verify this graph, then in “2” moves the marking of the PED node from child E to F.

2. In 3, CIS start from D and continue traversing until G, the end. VIS follows the markings, create “A-B-D-F-G”, then in “4”, change the marking for the New PED child node from B to C.

3. After 4 in CIS, continue from the newly marked node A forward until D, a split node, “But instead of marking, since D is already marked as being expanded, then it just stops there and does not mark anything. VIS then just follows the markings, deriving graph A-C-D-F-G. Since D has already been marked as expanded from step 2, it does not move any markings, and
since there are no more children of any PED nodes, it ends. The algorithm saves the creation of the last sub-path which is: A-C-D-E-G

If there is an OR split node whose instance subgraphs are free of errors, then the paths below this OR split node in later iterations are also free of structural errors and can be ignored in the sub-path creation step. Thus, the algorithm detects if a node has been expanded, and does not expand again. This condition saves the algorithm creating sub-path instances which have been checked before. The author in [PM07], calls this characteristic “deepest OR-split invariance property”. We do not need this property, because we always need to obtain all possible sub-paths.

5.2.1.4 Full sub-path Algorithm

After analyzing the previous traversal techniques and considering the needs we have in a traversal algorithm in order to satisfy our requirements we derive the full sub-path algorithm to derive all possible sub-paths.

We propose an algorithm for sub-path derivation that uses similar concepts from Depth First Search (DFS) [CLRS01] and M-S algorithms (MS) [PM07] to extract all the possible sub-paths in the workflow graph.

The full sub-path algorithm only differs from the MS create instance sub-path (CIS) algorithm in that our algorithm will create every sub-path instance possible.

In Fig. 5.4, our algorithm should not skip the creation of a sub-path, since in our work we need to obtain all possible sub-paths, so the optimizing condition in MS does not apply to our work. It could be said that our full sub path algorithm is a simplified version of the CIS (Create Instance Sub-path) part of the MS algorithm that does not check if a node has been expanded before.

In Fig. 5.5, a diagram of the steps to create all the sub-path instances for workflow graph in Fig. 4.1, in Fig. 5.3 we can see the steps taken to create the full sub graph in each iteration of the full sub graph algorithm:

Gray colored nodes mark the current sub-path being created. Nodes that make up the new segment traversed after following the next PED child node are dark grey and marked with a thick black dotted line.

1. The traversal moves forward as in DFS, remembering split nodes and its children for later traversal, until it reaches the end G. The path created is A-B-D-E-G. The next path will be the same as the previous until the current PED node which is D, resulting in: A-B-D.

2. The traversal continues as (1) with the next PED-node child: F until it reaches the end. The complete path created is A-B-D-F-G. The next path will be the same as the previous until the current PED node which is A, resulting in: A.

3. The traversal continues as in (1) with the next PED-node child: C until it reaches the end. The complete path created is A-C-D-E-G. The next path will be the same as the previous until the current PED node which is D, resulting in: A-C-D.

4. The traversal continues as in (1) with the next PED-node child: F until it reaches the end. The complete path created is A-C-D-F-G. The next path will be the same as the previous until the current PED node which is D, resulting in: A-C-D.
5. There are no more PED node children left, the traversal is over.

The algorithm derives all the possible sub-path instances of the workflow. In table 5.2 is a list of the first set of sub-path instances created for the workflow graph in Fig. 4.1. The other 2 sets of sub-paths are identical except that they start with the new PED-OR Nodes B and C.

**Table 5.2. First set of sub-paths under Full sub-path algorithm**

<table>
<thead>
<tr>
<th>Output of full sub-path algorithm</th>
<th>First set of Sub-Paths</th>
</tr>
</thead>
<tbody>
<tr>
<td>A → D → E → F</td>
<td></td>
</tr>
<tr>
<td>A → D → E → G</td>
<td></td>
</tr>
<tr>
<td>A → D → E → H</td>
<td></td>
</tr>
</tbody>
</table>

Each sub-path is complete since it represents a possible traversal of the graph from the start to the end node, and since we are expanding all children of every split-node. We are also guaranteed to obtain every sub-path possible because it explores all children of every split node and the graph is connected.
In table 5.2 we notice there are segments which are repeated (More on repeated segments and future work later). In this step, it could be deduced that the common or repeated segments should be eliminated, and just preserve the segments or nodes which are unique. We cannot avoid creating the sub-paths without including the common segments, since they give us a context and relative order for the execution of these services with respect to the instance sub-path. The sub-path also gives us the direct and indirect dependencies that these nodes have to other nodes. This list of subgraphs can be analyzed independently by any of the dependency algorithms available. We will see in the next section how each of these sub-paths is traversed and the dependency function is called with the relevant services obtained from these sub-paths.

In Fig. 5.6 the full-sub-path algorithm to derive all sub-paths is shown:

![Full sub-path algorithm](image)

**5.2.2 Dependency Detection**

The second major step of our algorithm refers to the invoking of a specific dependency function on the relevant activities from the list of sub-paths derived in step one. The algorithm checks if a dependency exists between these activities.
In Fig. 5.3 we can see all the major parts of the algorithm, including the derivation of sub-paths in the previous step. We can see that the second step of dependency detection has three sub-parts:

1. For each sub-path derived from the Full Sub-Path Algorithm:
2. While traversing each sub-path, call the dependency function with the relevant services as parameters,
   a) Dependency function performs dependency detection
   b) If there is a dependency between these services, create an entry on the dependency model.

The traversal of the sub-paths is independent from calling the dependency function, so they can be interchanged with another traversal or function, granted their interfaces are compatible. Modularizing each algorithm step will promote code re-use of both traversal and dependency function code in the implementation section.

Next we detail the dependency detection part of the algorithm

5.2.2.1 Sub-path traversal

This section deals with traversing the sub-paths created and choosing which services will be compared in order to derive possible dependencies. Depending on the dependency type, it will derive different candidate services. In the resource and time dependencies we describe, the services that will be selected by the sub-path traversal are pairs of adjacent or non-adjacent services. It could be the case with other dependency types that there need to be other sets of services derived, just as in the case of deriving dependencies that describe when one service depends on many services. These types of dependencies are classified as horizontal dependencies and are described in [MW09]. In the scope of our work we only deal with the case when one service depends on another service at one time. For resource dependencies, we derive adjacent and also indirect services as candidates between all sub-paths.

One issue concerning the traversal algorithm for resource dependencies is that it needs to call the dependency function in the logical order in which the services depend on each other.

First we define the semantics of deriving a pair of services H and C from the sub-path: as \( \{H, C\} \): “H depends on C”. This is a logical assumption since service H is at the end of the route which depends on C to deliver the correct pallets. On the other hand, the pair \( \{C, H\} \) not relevant to call the dependency function with, since C does not care what happens to H, C only delivers its pallets and does not depend on any other services to fulfill its resource SLA, so it would not be logical to state “C depends on H”, and thus is the sequence pair need not be derived. The algorithm needs to derive pairs of services along a sub-path starting from the last service and matching it against all other previous services and iterating to the service’s predecessor. The traversal approach is simple. Start at the end of the sub-path with service H and compare this service to all previous services in the sequence. Backtrack along the sub-path by obtaining H’s predecessor, compare again, successively until we reach the first service “C”. These resource dependency pairs are derived for service “F” and given to the resource dependency function as parameters: for the following path: \( \{C, D, E, H\} \)
1. \{H,E\} \{H,D\} \{H,C\}
2. \{E,D\}\{E,C\}
3. \{D,C\}

For time dependencies, we only need to derive adjacent pairs of services. Again, the ordering of activities within the pair is important. We order the successor node first, then its predecessor. The pairs for time dependency are only between adjacent activities.\footnote{For sub-path A-D-E-H, we would call the time dependency function with these pairs of services as parameters: \{H,E\} \{E,D\} \{D,A\} In Fig.\footnote{Fig.} 5.7, we show what pairs need to be derived for a) resources and b) time, the example is shown only for service “F”. The same procedure would be done for every service in the sub-path.}

An important issue to consider is that when the traversal algorithms analyze the sub-paths they will encounter many duplicate common segments, which will be repeatedly matched against a possible dependency. This can become a bottleneck specially if the dependency function needs to calculate a complex dependency. An example of this repetition of common segments can be seen in Fig\footnote{Fig.} 5.5 where the common segment “D-E” appears in all sub-paths which will be analyzed and matched against the dependency functions. A method to avoid calling the dependency function repeatedly with this same pair many times is with the use of an adjacency matrix. This matrix will record when a pair of services has already been used in a match, and avoid calling the dependency function if a repeated pair is detected. Creation of common sub-path segments and duplicate pairs is a limitation of the algorithm which will be discussed in chapter six\footnote{6.3.2}. As a contribution, an approach was made to
factor out common segments in sub-paths and therefore avoid duplicate pairs, but the approach proved too complex for practical use.

In this section, we have analyzed how we can traverse each sub-path in a dependency-specific way in order to derive relevant services. In the next section we will call dependency function with these services as parameters.

5.2.2.2 Dependency Detection

We have seen in chapter four in 4.2.1 that the task of dependency detection is domain-specific and requires knowledge about the workflow, its traversal and the SLA properties of the services. We have separated this functionality in two parts so that we can combine and substitute different traversal and detection algorithms. The traversal of the sub-paths described in the last section in 5.7 which also has its own logic to derive the relevant services, provides the dependency functions with the relevant services as parameters. The dependency functions can then focus on detecting the dependency without regard to the workflow or activity ordering. The dependency detection can use information from the sequence flow given in the previous sub-path traversal sub-step and/or from the SLA information for each service. The design of the dependency detection was made modular at this level in order to enable the substitution of independent dependency functions. The description of each function follows:

paragraphResource dependency function The resource dependency function takes two workflow activities as parameters, extracts the pallet id's of the output property of the first executing service, and matches it with the pallet id's of the input property of its successor service, and returns the set of common pallets matched. If there are no pallets in common, then there is no dependency between these two services. The traversal section in 5.7 guarantees that the order of the activities given as parameters is semantically correct, so that if we call the function with this parameters: “resource_dependency_function( H ,C )”, it means we are testing if activity H depends on activity C for resources. As discussed before, the traversal in 5.7 would not call the function with irrelevant parameters such as in: “resource_dependency_function( C ,H )” or “resource_dependency_function( A ,A )” which has no valid semantics for our dependency detection algorithm. Fig. 5.8 outlines the resource dependency algorithm:

Fig. 5.8. Pseudo-code for resource dependency function

paragraphTime dependency function The time dependency function takes two adjacent services based in the workflow sequence. As it has been explained
in the previous chapter in 4.1.2.2, the SLA of the services these potential time dependent services would then need to be verified, since the SLA information may be false. In this way, the SLA cannot guide the dependency detection, only the workflow sequence of the activities.

Fig. [5.9] outlines the time dependency algorithm:

```python
def time_dependency_function(dependent, antecedent):
    if antecedent.successor == dependent:
        return True
    else:
        return False
```

When checking for time dependencies, it could be the case that we are dealing with adjacent services, but their times inside the SLA are not correct. This would mean that there is an error in the entry of the SLA information, and the SLA for those services needs to be verified for correctness. The task of dependency detection lies at the core of our work. It compares and matches SLA properties of the services involved in the workflow. The detection algorithm is aided by the traversal algorithm which derives the relevant services to compare and in the order which the algorithm expects them. As a last step, the algorithm returns a value if there is a dependency between the services, the algorithm’s external interface records the information about the services, their properties, and dependency type.

5.2.2.3 Recording of Dependency Model Instance.

The recording of the dependency is the final step in the process of dependency detection. When the dependency detection returns a positive result for a dependency, a routine then adds each new found dependency into a dependency model. This dependency model was created as part of the TEXO project [H.W] and serves to model the dependencies between services and their properties.

The resulting model contains a reference to all the services in the workflow and one entry for each dependency. Each dependency references the services involved in the dependency. A brief excerpt of a dependency model instance shows a service dependency where service $E$ depends on $D$ for resource 5: The model also shows each service’s id as a key,

5.3 Conclusion

The approach to dependency detection presented in this chapter involves a two-step algorithm: First, it derives all possible sub-path instances. A next step takes these instances and derives the relevant services to be examined by a dependency function, and existing dependencies are recorded in the dependency model.
In the next chapter we will detail the technical implementation for these steps. We will see how the algorithm's characteristics fulfill the requirements set in the requirements chapter in 4.2.
Implementation and Evaluation

Chapter six discusses the technical realization of the dependency detection algorithm focusing on target platform integration, algorithm implementation and requirements evaluation. A description of the implementation is done, and the requirements seen in chapter four will be evaluated against the algorithm's functionality. Some limitations and constraints of the algorithm are also discussed in the conclusion sub-section 6.3.3.

6.1 Target Platform

First the overall architecture is examined, integration issues and platform requirements discussed.

6.1.0.4 TEXO-IOS and the ISE Development Workbench

In the scope of this thesis, the TEXO-IOS project is part of the Service Engineering work package under the TEXO-IOS project. The purpose of the Service Engineering work package is to model service and process descriptions under various perspectives.

In the context of the TEXO-IOS Architecture introduced in Fig 6.1, the dependency algorithm is implemented as a plug-in within the ISE Development Platform (ISE Workbench). The algorithm takes as input a workflow created under the ISE Workbench and interfaces with the SLA Manager discussed in sub-section 5.2.2.2 in order to obtain the SLA information. The output is a dependency model which will be used by the run-time and monitoring environment.

The following steps in 6.1 are taken by the components under the TEXO-IOS Architecture under the context of dependency derivation:

1. Design Time
   a) Provider creates service composition.
   b) Consumer Selects service.
   c) Negotiation via SLA Manager to obtain service SLAs.
   d) Detect dependencies using process and SLA descriptions.
   e) Output dependency model.
2. Run-Time
a) Run and monitor process.
b) If Process fails, service run-time uses dependency model to determine the affected services.

6.1.0.5 Integration into the target platform

The dependency detection plug-in resides inside the ISE Workbench, which is supported by a Eclipse Plug-in Development Environment. Delap defines a plug-in as:

"[…] structured bundles of code and/or data that contribute function to the system. Function can be contributed in the form of code libraries (Java classes with public [application program interfaces] APIs), platform extensions, or even documentation. Plug-ins can define extension
points, well-defined places where other plug-ins can add functionality. [...]” [Del06]

<table>
<thead>
<tr>
<th>Plugins</th>
</tr>
</thead>
<tbody>
<tr>
<td>USDL Editor</td>
</tr>
<tr>
<td>BPMN Modeler</td>
</tr>
<tr>
<td>Dependency Detection Plugin</td>
</tr>
<tr>
<td>Dependency Model Creator</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Systems</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISE Workbench</td>
</tr>
<tr>
<td>OSGI-Equinox</td>
</tr>
<tr>
<td>Java Runtime</td>
</tr>
</tbody>
</table>

![Fig. 6.2. ISE Workbench and plug-in Architecture](image)

The plugin created for this thesis provides functionality inside the ISE Workbench along other plug-ins, as illustrated in 6.2.

In order to integrate our plug-in in the TEXO platform it has these characteristics:

- The Integrated Design and Development Environment (ISE Workbench) focuses on the implementation of all relevant tools needed for the modeling and development of services and business processes.
- The plugin is integrated into the workbench via the Dependency Analysis tool accessible via Eclipse Menu and new Wizard inside the ISE Workbench.
- The plugin can load a BPMN process description previously defined inside the ISE workbench.
- A client interface in the dependency analysis plug-in connects to the SLA Manager and obtains all relevant SLA data [4.2].
- The plug-in creates an EMF dependency model description file, which is exported to the service run-time, where it will be used.

Our dependency algorithm is a plug-in which will analyze a BPMN description after its modeling via BPMN Editor tool.

The Eclipse Plug-In Architecture Eclipse under OSGI [Del06] was used to enable the collaboration and access to different plug-ins under one runtime environment. allows for Java projects with packages to be included. The
The implementation was done in the Java programming language. The SLA Manager is a web service created with JAXWS and JAXB API which is referenced via the dependency analysis SLA Client and provides the SLAs for each service via WS-AG formatted templates. The dependency model by [MW09] is used to create an EMF representation of the dependencies found.

The sequence and class diagrams below which show a high-level view of the different actors and classes collaborating in the module in order to perform the tasks of graph traversal, SLA Information retrieval, Dependency information matching and creation of a dependency model instance:

![Class diagram of dependency detection algorithm.](image)

The class diagram in Fig. 6.3 shows the major classes and their relationships involved in the algorithm. It follows the following steps.

1. The Main class calls the BPMN Model class to read the BPMN file description and convert it to an internal representation inside the BPMNModel class. The internal representation has BPMNElement objects inside representing the workflow.
2. The creation of the sub-graphs was implemented inside the BPMNModel class by applying the full sub-graph algorithm to the internal representation.

3. To obtain the SLA information from all the activities inside workflow, the SLAClient queries the SLA Manager described in 6.2.1.2.

4. Depending on which dependency is being analyzed, an appropriate Traversal object is instantiated. The Traversal class takes the sub-paths of the workflow and the domain-specific dependency algorithm as parameters. The corresponding sub-class of the Traversal traverses each sub-graph using its own dependency-specific method and calls the dependency-specific dependency algorithm with the relevant activities as parameters.

5. The Dependency algorithm takes the relevant activities involved in a potential dependency and their SLA information, depending on which dependency is being detected and applies the matching.

6. If a dependency is detected, a dependency with the information about each involved service is created by using the Dependency Model.

The sequence diagram also follows the steps and describes the major interactions done between the different actors 6.4.

6.2.1 Input Interfaces

First we describe how the algorithm obtains and pre-processes the necessary input information to detect dependencies.
6.2.1.1 BPMN Process Description

The algorithm can read a BPMN process description created by either the TEXO-BPMN Modeling Tool or by the normal Eclipse BPMN Tool provided under the Open source Eclipse Web-Tool-Package. (WTP). We use the XML Java API to parse the bpmn file. The BPMN file description defines a node to each BPMN element which can be an activity or an edge. An internal representation of these nodes is created inside the BPMNModel class which also has methods to obtain sub-graphs for the workflow, successor, predecessor and other utility functions.

6.2.1.2 SLA Manager

The dependency analysis plug-in performs retrieval of SLA information for each service via the external interface SLAClient to the SLA Manager and matches it to its corresponding service via its service key. Java types using JAXB Java API are used to marshal the information from the SLA Manager to the SLAClient inside the dependency analysis. The use of JAXB types allows every application domain such as logistics to define their own JAXB type hierarchy which will describe the specific information which will be accessed by the SLA Client. These types are then unmarshaled by the client plug-in for each service and queried like any normal Java object with getters and setters locally. Each service SLA is queried via its serviceKey obtained from the BPMNModel.

Fig. 6.5 shows some of the attributes which are part of an WS-Agreement. Sections of the agreement are converted into Java types and can be queried programatically inside the client. A part of the Java types is shown in Fig 6.6 and a typical usage of these types to query the properties of Logistics SLA properties inside our plug-in is shown in Fig 6.6.

```
...<ServiceKey> ServiceK1 </ServiceKey>
...

<logistics>
  <resource>
    <pallets>1 2 3 4 5 6</pallets>
  </resource>
  <pickUpTime>22.4.2009 7:00</pickUpTime>
  <deliveryTime>22.4.2009 9:00</deliveryTime>
</logistics>
```

Fig. 6.5. Logistics schema

When the remote SLA Manager service has been successfully queried, Logistics Java Object for each service in the composition is unmarshaled locally, and it can then be used to compare different service’s SLAs.
public class Logistics
String serviceKey;
String
resources;
Date pickUpTime;
Date deliveryTime;
...

Logistics myLogisticsInfo1=Agreement.getAgreementByKey
("SvcKey1").getLogistics;

Logistics myLogisticsInfo2=Agreement.getAgreementByKey
("SvcKey2").getLogistics;

myResource1s=myLogisticsInfo1.getResource();
myPickUpTime1=myLogisticsInfo1.getPickupTime();

myResource2s=myLogisticsInfo2.getResource();
myPickUpTime2=myLogisticsInfo2.getPickupTime();
myDeliveryTime2=myLogisticsInfo2.getDeliveryTime();

//Sample matching of two service properties
if (myResource1s.contains(myResource2s))
//There exist common resources between both services

Fig. 6.6. Logistics type and usage

6.2.2 Output interface

The dependency model was created in EMF to allow for a standard modeling and interchange format. Eclipse EMF allows for the creation and edition of metamodels and their instances using an open model which can be transformed into other models such as xsd.

The main class is responsible for initializing the dependency model at the beginning of the dependency detection. The dependency model allows for creation of dependencies with cardinality, dependency types and relations to the services to which it relates. The dependency model used is described in [MW09], and follows works such as [Tol03] that also define dependencies as a class model. The model provides concrete classes which describe the dependency relations between different activities. Here is a class diagram of the EMF model: We can see the EMF Entities representing a dependency and the related dependant and antecedent services.
Fig. 6.7. EMF Service Dependency Model

Each Traversal of the workflow will add the corresponding dependencies corresponding to its dependency type. After all the traversals for each dependency type have added the dependencies, the main class will then output this model to an external file with .xmi format. XMI is an xml formatted metamodel instance description used in Eclipse framework to work with EMF. A sample output dependency follows in the following code segment:

```xml
<dep:service serviceName="Truck DD" serviceKey="Key1" />
<dep:service serviceName="TGL Truck" serviceKey="Key2" />
<dep:resourceDependency>
  <dependant>tdd</dependant>
  <antecedent>tgl</antecedent>
  <dep:resourceIDList>
    <dep:resourceID>P1</dep:resourceID>
    <dep:resourceID>P2</dep:resourceID>
  </dep:resourceIDList>
</dep:resourceDependency>
```

Fig. 6.8. Logistics schema based on [MW09]

6.3 Evaluation

We have evaluated the algorithm with regard to the requirements set in chapter four. Here we describe how each requirement is fulfilled.

6.3.1 Evaluating Requirements

This section refers to section 4.2 which outlines the requirements that our module should fulfill in the areas of dependency algorithms and platform.
6.3.1.1 Dependency Function requirements

*RDR1/TDR1: Order of activities is resolved by the algorithm:*

This requirement is fulfilled because the resource dependency traversal resolves activities in sequentially order matching adjacent and non-adjacent services. The time dependency algorithm only derives service pairs following a sequential order. These requirements are fulfilled by the Resource and Time Traversal algorithm and how it obtains the specific services to compare. The concept of the pair derivation is explained in 5.7.

*RDR2/TDR2: Dependencies occur between pairs of services.*

This requirement is fulfilled because the dependencies are detected by the specific traversal according to its dependency type 5.7.

*RDR32: The algorithm detects and matches subsets of matching resources in any activity along the route.*

This requirement is partially fulfilled because every service in all sub-paths are examined for potential dependencies. The output of a service is matched against the input of another 5.8.

6.3.1.2 Platform requirements

The requirement of platform integration involves many sub-requirements so that the solution works within the TEXO-ISE Workbench Environment. Next we consider the platform requirements which are specific to the Java project which conforms the plug-in. These requirements are also important since they will enable the plug-in to be extended and change with new requirements.

*PR1: The algorithm should be integrated into the platform.*

This requirement is discussed in 6.1.0.5.

*PR2: The algorithm performs its different tasks via interchangeable modules.*

Modularity is fulfilled in the way that sub-parts of the algorithm can be substituted by others and can work together. The design of the time and resource dependency algorithms is encapsulated in separate functions to allow creation of independent and interchangeable dependency algorithms 6.3. This is evident in the design of the Traversal and DependencyFunction classes, which can be one of many concrete traversal and dependency function sub classes, as can be seen in the class diagram in figure 6.3.

*PR3: Extensibility enables the algorithm to grow its functionality.*

Extensibility is fulfilled because the input interface accepts BPMN description files created in the TEXO BPMN Modeler Tool or the Eclipse open-source BPMN Tool. This was a necessary extension because the ISE Workbench TEXO BPMN Modeler Tool is not practical to use during development, since we need
a re-build of the whole Workbench. On the other hand, the Eclipse open-source BPMN Tool is available during code development all the time. Further extensibility could be to allow the parsing of different types of WS-Agreement formats, or the plugging in of new dependency algorithms via a rule engine or dependency repository. Presently, the extensibility provided is that of the ability of coding new traversal and dependency algorithms independently as subclasses of the Traversal and DependencyFunction abstract classes, and observing their interfaces.

**PR4: The algorithm should support processes with an increasing number of activities.**

Scalability is fulfilled because the algorithm successfully handles processes of varying sizes and complexities. The evaluation has section evaluated many processes and it has successfully examined them within an appropriate time.

### 6.3.2 Performance evaluation

In [Car06], the author measures various complexity metrics in regard to workflows. He considers two dimensions for evaluating workflow complexity which are its structure, activity, control and data flow complexity. The workflows in our thesis are only consider AND-splits, and do not associate any data-flow directly to them so a complexity. Other works consider the sum of number of splits and the children nodes at each split as a measure of complexity [Maw04].

A relevant metric for this evaluation that is discussed in [Car06] and is the Control-flow complexity (CFC). It is associated to the number of splits needed to describe the workflow. It sums the number of children in each split. This is calculated via the formula:

\[
CFC(P) = \sum_{i \in \text{AND-splits of } P} CFC_{\text{AND-split}}(k) + \\
\sum_{i \in \text{XOR-splits of } P} CFC_{\text{XOR-split}}(i) + \\
\sum_{i \in \text{XOR-splits of } P} CFC_{\text{XOR-split}}(i)
\]

![Fig. 6.9. Control-flow complexity (CFC) Formula by [Car06].](image)

A similar upper bound has been derived as the sum of the product of child nodes of each of the split nodes in [PM07], which coincides with also workflow structure. The complexity of the algorithm is related to the number of children at each split node and the number of nodes at each level of the graph.
6.3.2.1 Evaluation metrics and processes

In this thesis, a metric that we use to measure the performance of the algorithm is the time to complete particular steps of the algorithm. In order to derive other metrics, we apply various steps of the algorithm and derive various measures such as number of pairs, dependencies and duplicates processed by the algorithm when deriving the relevant dependencies. These time and measurement metrics will give us an idea of what percentage of time is taken by each of the steps of the algorithm. We can examine which steps produce more output and take more time than others.

Relevant algorithm attributes to be measured are described next:

1. The runtime needed to obtain different number of sub-paths created is measured. This gives us an idea of the amount of processing needed by the algorithm to analyze all of the sub-paths and derive the dependencies.
2. The time needed to derive different number of pairs for dependency evaluation provides the amount of processing needed in the first step of the algorithm.
3. When obtaining the pairs many of the pairs are duplicated and need to be checked. The difference of time between obtaining pairs and obtaining duplicates is measured to measure the effective work and duplicate work done by the algorithm.
4. By subtracting the pairs and the duplicates, we can get the number of pairs without duplicates that will actually be used by the dependency function as parameters. As we have seen in the concept section in [6.2] the existence of duplicates in the sub-paths is a limitation of the algorithm, and it is important to measure this extra processing. The difference between duplicates and the pairs that will be used in the final dependency detection gives an idea of the overall performance of the overall approach to obtain dependencies.
5. Another important measure is the number of dependencies found in each dependency function. This measure depends on the particular SLA of each service or how the services are structured inside the process. In the case of time dependencies the number of time pairs and dependencies is the same, because a time dependency exists between a pair of adjacent services in a sub-path, which are the same pairs derived from the traversal. The dependencies derived from the resource dependency algorithm are more difficult to obtain since they require that for each service in the process an SLA be configured and that all these SLAs correspond to a valid composition of SLAs. (i.e, the dependencies of the services they represent are correctly configured between all services, such as in [4.1]. In the scope of this thesis, we have derived correctly configured SLAs for the Logistics sample process in [4.1]. From the Logistics process we have derived the times to get its resource dependencies. There is no further data available about resource dependencies for the other processes.
6. As a final measurement, the total processing time has been measured, which includes loading and parsing of the process obtaining paths, pairs, dependencies until the recording of the dependency model. This total time represents a complete iteration of the algorithm that could be expected under a typical usage of the system.

67
The results of these measures will be analyzed in graphs in the evaluation results section in [6.3.2.3]. In the next section we discuss the procedure to evaluate these measures to ensure reliable results.

### 6.3.2.2 Evaluation procedure

The tests conducted were made on a Pentium IV, 3.20Ghz With 2Gb RAM. 60 Processes were running in the machine, which was the minimum of processes that could be run in the machine. The algorithm was run as a Java project under the Eclipse 3.3 platform, running a Java 1.6 Virtual Machine.

The tests were conducted by measuring the attributes in the metrics section in [6.3.2.1] for each of the six business processes described. The tests were iterated five-hundred times and the average value was obtained. The results were then graphed to obtain the relevant measurements.

**Processes complexity**

In order to test the algorithm in this thesis, we looked for processes with an increasing complexity. The criteria to measure the complexity of a process was taken from [Car06] and [Maw04]. We took complexity measures of a process such as the number of nodes, number of split-join combinations and number of sub-paths which can be created from the process. All these factors affect process complexity, so we have chosen various processes with a different combination of nodes paths, nodes and split nodes. As concrete examples we use the processes which have been tested by Mendling in [Men08]. They represent processes from various fields of knowledge which provide a realistic reference. Other processes were designed with a structured number of splits, joins and children at each split based on the complexity analysis of [Maw04]. These are basically graphs with growing number of children at each of their split nodes. The sample process from Fig.4.1 is also tested, which is the basic process we have used throughout the thesis.

Table 6.1 shows all processes used in the evaluation and their static characteristics:

### 6.3.2.3 Evaluation results

Table 6.2 shows the time needed to obtain different metrics:
Table 6.3 shows the number of pairs generated for processes with an increasing number of sub-paths:

In this section different time measurements of the algorithm are compared and interpreted.

The graph in Fig.6.10 measures the running time to obtain a different number of paths for the dependency detection process. We can see that the time to obtain the sub-paths remains linear with respect to the time, and we will see in later analysis that it incurs a decreasing amount of time with regard to other steps:

The second step in the algorithm derives the time and resource pairs that will be used in the dependency detection. The graph in Fig.6.11 shows
Table 6.1. Example Processes.

<table>
<thead>
<tr>
<th>Name-Ref</th>
<th>Description</th>
<th>Nodes</th>
<th>Splits</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ex14.1</td>
<td>Logistics Sample</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>A60 [Men08]</td>
<td>Sales-QM</td>
<td>38</td>
<td>5</td>
</tr>
<tr>
<td>Ex2 [Maw04]</td>
<td>3 levels, 4 child.p.split</td>
<td>103</td>
<td>8</td>
</tr>
<tr>
<td>Ex3 [Maw04]</td>
<td>4 levels, 4 children p.split</td>
<td>102</td>
<td>8</td>
</tr>
<tr>
<td>A58 [Men08]</td>
<td>Revenue-cost control</td>
<td>53</td>
<td>12</td>
</tr>
<tr>
<td>A71 [Men08]</td>
<td>Production- QM</td>
<td>75</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 6.2. Time to get different measurements depending on number of paths(Ms)

<table>
<thead>
<tr>
<th>Sub-paths:</th>
<th>9</th>
<th>66</th>
<th>256</th>
<th>469</th>
<th>928</th>
<th>1270</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time: Subpaths:</td>
<td>0.0003</td>
<td>0.0009</td>
<td>0.0050</td>
<td>0.0078</td>
<td>0.0224</td>
<td>0.0426</td>
</tr>
<tr>
<td>Time for Resource pairs:</td>
<td>0.0005</td>
<td>0.0086</td>
<td>0.0966</td>
<td>0.1360</td>
<td>1.0854</td>
<td>1.5463</td>
</tr>
<tr>
<td>Time for Time Pairs</td>
<td>0.0007</td>
<td>0.0073</td>
<td>0.0646</td>
<td>0.1026</td>
<td>0.3458</td>
<td>0.5465</td>
</tr>
<tr>
<td>Time for Time Dependencies</td>
<td>3.8072</td>
<td>9.1869</td>
<td>66.5932</td>
<td>103.5194</td>
<td>658.2060</td>
<td>1011.7485</td>
</tr>
<tr>
<td>Total Time</td>
<td>4.9503</td>
<td>9.3752</td>
<td>67.2997</td>
<td>104.1560</td>
<td>659.8847</td>
<td>1014.1807</td>
</tr>
</tbody>
</table>

Table 6.3. Number of pairs vs number of sub-paths

<table>
<thead>
<tr>
<th>Sub-paths:</th>
<th>9</th>
<th>66</th>
<th>256</th>
<th>469</th>
<th>928</th>
<th>1270</th>
</tr>
</thead>
<tbody>
<tr>
<td>Resource pairs:</td>
<td>32</td>
<td>1045</td>
<td>8874</td>
<td>12055</td>
<td>140403</td>
<td>178858</td>
</tr>
<tr>
<td>-without duplicates:</td>
<td>22</td>
<td>216</td>
<td>342</td>
<td>1077</td>
<td>993</td>
<td>1484</td>
</tr>
<tr>
<td>Time pairs:</td>
<td>27</td>
<td>329</td>
<td>2048</td>
<td>3283</td>
<td>15476</td>
<td>20652</td>
</tr>
<tr>
<td>-without duplicates:</td>
<td>7</td>
<td>38</td>
<td>50</td>
<td>165</td>
<td>61</td>
<td>98</td>
</tr>
</tbody>
</table>

the time needed to obtain the resource dependency pairs and 6.12 shows the time needed to obtain the time dependency pairs. There is a non-linear relation between the amount of time measured to derive the pairs. The time to derive a higher amount of pairs in 6.11 and 6.12 grows non-linearly and we can expect to take a higher time to derive a higher number of pairs.

Fig 6.13 shows the total time needed to obtain time dependencies. Resource dependencies could not be obtained 6.3.2.1.

In this measurement we see a proportionately higher amount of total time it takes to obtain time dependencies compared to the rest of the measurements. An example is that it takes 1200Ms to obtain and record time dependencies in Fig. 6.13 against only .6 Ms to derive the time dependency pairs in Fig. 6.12. This large difference can be due to extra process-
ing due to the mapping of the dependency information into EMF objects, creation and saving the dependency model into a file. Nevertheless, this finding shows that the total time to obtain time dependencies is much larger than the previous steps. The only added processing was the recording of the dependency model, so we can conclude that this last step incurs the most time. Further testing obtaining the total time for resource dependencies is required for a more relevant comparison. Unfortunately, resource dependencies could not be obtained.

Fig. 6.14 shows the difference between the number of time dependency pairs including duplicates and number of dependency pairs without duplicates obtained with a growing amount of sub-graphs as input. In this measurement we see a proportionately higher number of duplicate pairs is created with the growing number of paths in a process. We can see that as the processes grow in complexity, the number of duplicates grows, and the actual number of dependencies remains linear. This points to the fact that duplicates could become a factor for diminished performance. The difference between resource pairs and resource pairs duplicates is similar.

In Fig. 6.15 we analyze the different steps of the algorithm and the percentage of time it takes to perform each function compared to the total time it takes to run the individual steps. Fig. 6.15 shows that with the in-
crease of process complexity, the percentage of time used in the creation of the resource dependency pairs also increases. The percentage of time to get the paths and to load the model becomes less and less with growing number of sub-paths. The reasons for these results could be that in order to create the time dependency pairs it only requires the traversal of the path one time in a linear way, whereas the creation of the resource dependency pairs requires the combination of every service against all of its other preceding services, making this an exponential increase in processing time. From the graphs examined, we can see that the steps of the algorithm that take the most time are the calculation of the pairs to call the dependency functions. Inside the calculation of getting the pairs, we have identified that the time it takes to process duplicate pairs for both time and resource dependencies takes most of the time, and it grows exponentially as the process grows more complex. The other measures such as loading the model and getting the paths remain stable across the processes. We can expect that with growing complexity of a process can see a non-linear rise in time to calculate the pairs, and also the processing of the dependency model. Another finding was the total time calculation to get time dependencies. This time was much higher than all of the other measurements. The only added processing when deriving this total time was the recording of the dependency model, and further testing would need
to be done when deriving resource dependencies to obtain more relevant results. In conclusion, the overall times obtained showed a performance which is acceptable under the design-time requirements. The highest time derived was of 1.2Ms for the most number sub-graphs, so the algorithm’s performance appears acceptable.

6.3.3 Limitations

In this thesis, we have devised an algorithm that incorporates a common sub-path derivation method and specific traversal and detection algorithms for each dependency type. The creation of linear sub-paths is a convenient way to find linear dependencies. Then the algorithm examines all paths, and obtain all dependencies for the workflow. This approach is modular and accommodates the dependencies which we have examined, but the derivation of relevant pairs produces a large amount of duplicates, as can be seen in the performance evaluation. With larger processes this derivation of pairs could become a performance bottleneck.

Another limitation is the fact that resource dependencies are implicit in the SLA (i.e., pickup time and delivery time of a service), and if the SLA information is incorrectly defined in the SLA, the dependency is no longer valid or needs to be verified for validity. For time dependencies, this is not
the case since they are only based on the workflow information. Neverthe-
less an SLA verification phase needs to be incorporated to add reliability in
the information provided and avoid mismatches between sequence flow
and SLA information.
Anther limitation of the algorithm is that it does not support cyclic work-
flows or sub-processes. This would require different traversal logic for the
algorithm and more levels of abstraction for the internal representation
of the workflow. Other dependency types which are not supported are
vertical dependencies which require the creation of a new traversal and
dependency detection algorithm which is not implemented in this thesis.
The last chapter concludes the thesis by summarizing the main points and
providing a future outlook for the area of service dependencies in service
compositions using SLAs.
Fig. 6.14. Difference between pairs and duplicate pairs in time dependencies.
Fig. 6.15. Percentage of time in steps of the dependency detection algorithm
Chapter seven presents the summary and outlook for the thesis.

7.1 Summary

In this thesis, we have presented our approach to detect dependencies within a business process using a BPMN notation [OMG08] and its SLAs [And07]. After analyzing related work in the area, we have defined resource and time dependencies in the area of logistics. The algorithm that implements the approach is integrated into the ISE Workbench which allows it to collaborate with other service descriptions and process modeling tools [ea09]. The implementation was made modular so it can incorporate the detection of new dependency types by analyzing their sequence flow and different service properties in separate steps. In order to validate the implementation, we have created sample workflows with increasing complexity. The evaluation of such sample processes has exposed performance issues when obtaining the relevant pairs of services. As a contribution, an approach was made to analyze and factor out these common segments in sub-paths which lead to duplicate pairs, although the approach proved too complex for practical use.

In comparison to the related work, our approach most closely resembles [BWRJ08] in the logic of modeling dependencies, although in our work we derive different types of dependencies and we also consider interservice dependencies, not just dependencies between atomic services and its composition. Other promising approaches make use of semantic frameworks such as in [ZBH08] or use Description logic rules [Aga07], but they fall out of scope of this thesis. Overall, our approach is inspired by concepts from related work, but its modular implementation, use of SLAs and integration into a service development environment make it unique ([ZBH08], [BWRJ08], [LA05], [And07]).

7.2 Future Work

Future work could include the use of dependency ontologies, logical rules or semantic annotations to define and process dependency detection. The
promise of an integrated semantic technology for service-oriented systems is described by [LA05], [Tol03] and [ZBH08]. We should always consider issues such as the feasible integration into the current platform and compatibility of its components. In the case of this thesis, it required the algorithm to integrate with the platform, and the approach followed these requirements.

Other future work includes the discovery of different dependency types such as ones dealing with specific order of operations of certain activities in a process is a dependency type which has been described in [ADW08]. Parallel time dependencies such as in [QS06] could be supported in order to model complex simultaneity constraints between services, which can occur in may areas. Further work could be done with the handling of more complex types of processes such as cyclic flows, or flows with subprocesses. In order to accommodate more complex process constructs would require the re-structuring of traversal logic in the whole algorithm since the algorithm only supports the creation of linear paths.

In the context of TEXO-IOS we can considering scenarios in other areas can be derived in order to examine new dependencies and test the current ones we have in these scenarios, and how they can be extended, improved, refactored or re-used.


H.W. Walter Mattauch H.Weber. The thesesus research program on technologies for the internet of services.


Mil06. Bill Miller. Bpm and soa are natural allies, not enemies. webservices.org, 2006.


