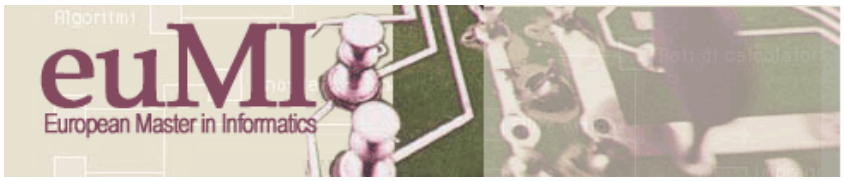

UNIVERSITA' DEGLI STUDI DI TRENTO

Facoltà di Scienze Matematiche, Fisiche e Naturali

Corso di Laurea Specialistica in Informatica
within European Master in Informatics

Final Thesis

Workflow Meets Peer to Peer: A Taverna Interface to LCC

Relatore/1st Reader:

Name: Prof. Maurizio Marchese
Institution: University of Trento, Italy.

ControRelatore/ 2nd Reader:

Name: Prof. Dave Robertson
Institution: The University of Edinburgh, UK.

Laureando/Graduant:

Name: Muhammad Hassan Akram

Anno Accademico 2006 - 2007

Abstract

Grid and agent based technologies have progressed significantly over time. Growing interest in the field of scientific workflows has led to the development of several workflow editing and enactment systems that are targeted towards scientists having little knowledge about the complexities of the world of Grid workflows and architecture. The peer-to-peer agent world offers the benefit of developing software components that are able to operate in open, dynamic and uncertain environments while making local decisions at runtime. There is now a growing interest of the scientific community to converge these existing technologies together in order to take advantage of the benefits offered by agent based peer-to-peer systems.

The Taverna system is a workflow editing and enactment system with graphical interfaces to build and execute the scientific workflows. The Taverna system uses an internal language (SCUFL) to define scientific workflows. It relies on a centralized way to define and execute the workflows and it has never been applied to a peer-to-peer system. We take up this work in order to show that workflow definition language SCUFL can also be modeled through Lightweight Coordination Calculus (LCC) language which can be used in the peer-to-peer knowledge sharing environment produced by the *OpenKnowledge* project.

This work further bridges the gap between existing grid technologies and agent based peer-to-peer technologies by showing that an existing scientific workflow system language (SCUFL) can be modeled and translated to an interaction model definition language (LCC) for multi agent peer-to-peer systems.

Contents

Chapter 1

Introduction

We will highlight briefly what grid and agent technologies are and how they have progressed. We will also highlight how with the passage of time there is a push to converge these existing technologies.

1.1 Technology Overview

1.1.1 Grid Technology

Advances in the field of information technology revolutionize how knowledge sharing and scientific experiments are conducted in other fields of science such as bioinformatics, cheminformatics, ecoinformatics, and geoinformatics etc. As the internet brought along many ways for users to share knowledge, the advancement in information technology produced the Grid to allow users to share computing power. The potential benefits of this advancement to businesses, scientists and governments were great enough to justify the development of Grid infrastructure and applications.[1][2]

With the development of Grid infrastructure the scientific communities are able to process large datasets, run and share their experiments and findings. Applications written over the Grid infrastructure were made so that they can integrate autonomous services over the web. With the development in service-oriented architecture for the web, it was possible to compose and execute experiments by using these collections of services. The scientists could now compose a scientific workflow over these autonomous services and execute their experiments.[3]

With the huge amount of distributed web components and services becoming available, manual discovery and integration of these resources to the workflow was not considered to be a realistic solution. In such a scenario the value of formalizing more of the semantics of a resource, as proposed

for the semantic web, was advocated for the grid to form *Semantic Grid*. This led to studies as to how distributed resources could be discovered and integrated in an automated way based on their semantics and syntax into a Grid workflow. [3][4]

The growing interest in the field of scientific workflows has lead to the development of several workflow editing and enactment systems that are targeted toward scientists having little knowledge about the complexities of the world of Grid workflows and architecture. This growing interest is evident from several recent workshops as highlighted by Ludascher et al in [2]. Some of the known scientific workflow systems that were recently developed are Kepler [2], Taverna [6] and the Triana [7].

These scientific workflow editing and enactment systems are targeted toward scientists having little knowledge about the complexities of the world of Grid workflows and architecture. The Taverna system enables workflow editing and enactment through its graphical interfaces to build and execute the workflow. The Taverna system is not intended as programming language but it still relies on an internal tool language, SCUFL, to define the workflow which is intended to remain hidden from the users. It is especially useful to bioinformaticians who can easily use and integrate molecular biology tools and databases that are becoming increasingly available on the web through their web interfaces and especially through web services. Taverna system thus allows users to easily define high level workflows by combining different resources to conduct their analysis.[5]

1.1.2 Agents

An agent is a "computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives" [8]. Jennings [9] elaborates this definition and explains that agents are problem-solving entities with well-defined boundaries and interfaces. They take inputs through their environment sensors and produce their effects/outputs in order to achieve a certain objective. Being autonomous means that the agents have control over their internal state and they can behave in a flexible problem-solving manner in order to fulfill their objectives. Agent can be reactive (replying to the changes in its environment), proactive (exhibit a goal directed behavior and take initiatives accordingly) or social (able to interact with other agents in order to satisfy personal agenda) [8].

A multiagent system is a system in which multiple agents act on behalf of their agencies with different goals and motivations. The agents will be

required to cooperate, coordinate and negotiate with each other. In multiagent system the participating agents must conform to an interaction model using a certain dialog protocol. In an interaction model an agent can take up a certain role in the interaction which determines the rights, duties and opportunities of the agent. The role thus limits the possible actions that can be taken up by the agent during the interaction. Policies can also be defined to formulate the acceptable behaviors or social norms for the participating agents. The dialog protocols define how and under what circumstances the interaction between the agents can take place. It also describes the order and type of messages that are passed between the agents. This is highlighted by Robertson et al in [10]. We must emphasize here that the dialog protocols do not define low level communication details but rather specify high level communication issues that (partially) specify how an agent should rationalize.

1.1.3 Grids and Agents

A few years back a small part of the scientific community began to converge grid based technology and agent based peer-to-peer technology. Grid technology initially primarily focused on how to reliably compose the distributed components across the internet for executing a workflow. This resulted in the development of frameworks and toolkits to allow resource sharing and coordinated problem sharing in dynamic environment. It was now possible for users to create virtual communities where the resources were distributed over the network but where each resource operates as a community member to achieve a common goal. [11][12]

While grid technology was more focused on developing frameworks, the agent community was focused on coming up with agent based peer-to-peer technologies. The main focus was to come up with algorithms for agents to do reasoning, to adopt social behaviors and interaction roles, to flexibly solve problems using defined interfaces and all this was achieved while keeping the autonomity of the agent in perspective. The agent community did produce such algorithms and techniques but did not establish practical platforms to support such architectures. [11][12]

It is now clear that grid based software needs to adopt features such as flexibility and distributed decision making from the agent community. The agent community needs to come up with standard distributed platforms for their aspirations to be realized. Initiatives to achieve these objectives are underway and the *OpenKnowledge* project is a recent example of this [13].

As mentioned earlier there is increasing interest within the scientific community to converge agent and peer-to-peer technology with grid technology. This is evident from the recent series of workshops [14] [15] and a new journal [16]. Although there is a push to integrate features of agent domain into the grid domain, there has been little effort to actually transform the existing widely used workflow languages and tools into the agent coordination languages. Our work in this respect would be novel as we aim to translate one of the more commonly used languages *SCUFL* and the associated tool *Taverna* into Lightweight Coordination Calculus (LCC), which defines the interaction models for the participating peers in a peer-to-peer manner.

1.2 Problem & Method

For the past few years we have seen an increasing interest in the scientific community to converge the more centralized grid world with the more flexible world of multi-agent peer to peer system world [11]. The motivation for doing this is clearly to take advantage of the benefits offered by the peer-to-peer world. Peer-to-peer agents are autonomous and flexible software components that are designed to operate in an open, dynamic, uncertain environment making decisions at runtime.

The Taverna project provides a language (SCUFL) and a software tool to build scientific workflows by using distributed components over the Internet. But as Taverna relies on a centralized way to define and execute these workflows, it has never been applied to a peer-to-peer environment. Our work in this context will further bridge this gap. We will show that the existing grid workflow building language (SCUFL) can also be modeled through the Lightweight Coordination Calculus (LCC) language which can directly be used in the peer-to-peer knowledge sharing environment produced by the *OpenKnowledge* project.

The purpose of this project is thus to compare Taverna with LCC. We take up this project with the basic hypothesis that the LCC language can be used to serve the same purpose as the Taverna's underlying workflow definition language SCUFL, so workflows that can be defined through SCUFL language can also be defined effectively through LCC. The project will aim to evaluate this hypothesis and based on the evaluation results would propose a translator to map the SCUFL language onto the LCC language.

We will start our analysis by comparing Taverna with LCC. In this regard we will examine how different workflow characteristics expressed in Taverna can also be expressed in LCC while keeping their functionality intact. The internal workflow language for Taverna (SCUFL) will be further analyzed

to evaluate the function of different language elements and also to see how different language elements can be generally transformed into LCC.

To further evaluate our hypothesis we will suggest a system of rewrite rules to do the translation between the SCUFL language and the LCC language. This system of rewrite rules will try to specify how the workflow elements in SCUFL will be extracted and reused in specifying the workflow interaction models in LCC language. The system of rewrite rules would also specify how implicit rules to specify the data and control flows between distributed components in SCUFL language will be mapped into interaction messages and clauses in LCC.

The LCC obtained from translation phase can be executed by using any one of the following two methods. The first method for running the obtained LCC interaction models is to use the Java based *OpenKnowledge* kernel which is currently under development. The other method for execution is to simulate LCC interaction models by using the *OpenKnowledge* Prolog interpreter. We will simulate the LCC interaction models obtained using the Prolog interpreter to conduct our analysis.

1.3 Related Work

The work conducted in this thesis is similar in nature to the one done by Guo et al in [17], which proposes enacting distributed workflow using BPEL4WS [18] on a multi-agent platform. This business process execution language (BPEL4WS) is an XML based language that is run by a centralized workflow engine. Guo et al [17] were able to produce an LCC protocol for multiple agents that could be used to interpret and enact existing BPEL4WS workflows.

BPEL4WS is an imperative language and it defines business processes through a set of *activities* each representing a different concept in the process. The basic activities (i.e. Recieve, Reply, Invoke, Assign, Throw etc) define how interaction with the webservices will be carried out while the structural activities (i.e. Sequence, Switch, While, Pick, flow etc) define how other activities will be composed and bound together. BPEL4WS thus offers a sophisticated and an elaborate set of constructs to authors to define the business process workflows. SCUFL on the other hand does not offer such programming constructs and defines simple graph structured data flows. There are no explicit structural constructs in SCUFL and it is only able to define iteration strategy over the inputs for a processing element in the workflow. It also allows processing elements with mismatching data types to be coupled together and thus it is easier for authors to build SCUFL

workflows. BPEL4WS offers constructs for fault handling and asynchronous message passing which SCUFL lacks. We will present a detailed overview of SCUFL language in the following Background chapter.[18][19]

Our approach of converting SCUFL to LCC differs from the one taken in [17], as we are adopting the most straight forward approach of mapping the SCUFL language elements to LCC protocol. Our decision to pursue this is based on the inherent difference between BPEL4WS and SCUFL workflow languages. BPEL4WS is an imperative language that is able to define control-flow, asynchronous messaging, error and fault handling to define a business workflow. SCUFL is comparatively a simpler language with a simple control-flow structure and is able to represent simple graph-structured scientific workflow language. Guo et al [17] noted that conducting the language mapping of BPEL4WS to LCC will not be possible in all circumstance due to its imperative nature. We think that the SCUFL language on the other hand due to its simpler structure can be mapped onto LCC directly.

1.4 Thesis Structure

In the second chapter, *Background*, we will take a detailed look at the SCUFL language and the LCC language. The *Algorithm* chapter will highlight the proposed translation algorithms for mapping SCUFL into LCC interaction protocols. We will then explain how simulations for the obtained LCC interaction models can be carried out in *Experiment* chapter. In the *Evaluation* chapter we analyze and evaluate our work and findings. We then summarize our findings in the *Conclusion* chapter.

Chapter 2

Background

We will now take a detailed look at the specifications of the Lightweight Coordination Calculus (LCC) and the Simple Conceptual Unified Flow Language (SCUFL).

2.1 Lightweight Coordination Calculus (LCC)

2.1.1 What is LCC?

Lightweight Coordination Calculus (LCC) is process calculus for specifying multi-agent coordination. Multi-agent systems in general can be viewed as state based systems where agents change their state and roles while interacting with each other. Each agent participating in an interaction scene must conform to certain conventions, rules and laws (i.e. social norms) to ensure the integrity of the coordination. The social norms in LCC can be defined as message passing behaviors that are associated with specific agent roles. This process calculus can be used to describe social norms of a state based agent system in a completely distributed manner without any central control and thus is suitable for peer-to-peer environment.[23][21]

The interaction between multiple agents is done through message passing but what messages are actually communicated or anticipated by an agent in the system are governed by some law or norm. So an agent while communicating should conform to some sort of interaction role that describes the constraints and conditions under which a certain communication can be carried out. The right order of communication messages in the system among the agent ensures that expected behavior of the system is carried out. One perspective of LCC is to view it as temporal since it imposes partial ordering on message passing between the agents as highlighted by [21]. This view of LCC is more relevant in context to our work of translating a workflow language into LCC.

The language defined by the LCC is declarative in the sense that it is defined independent of any execution architecture. The language is similar to logic programming languages and is similar to a Horn clause format for defining a protocol. An agent role in this context can be viewed as the head of the clause and the body would then contain the definition of agent's role behavior.

2.1.2 LCC Syntax

We will now describe the syntax for the LCC language used to describe agent communication protocols. An agent communication framework can be defined as a set of clauses where a clause is an agent's role definition. As mentioned earlier the agent interaction is done through message passing. This message passing can be based on fulfillment of certain constraint and similarly receiving a message might imply a certain constraint. The constraints mentioned can be defined in any language that is supported by the constraint solver used. We will describe the constraints in the form of first order predicate calculus that carry terms in the same structured format as the Prolog and can be formulated by using conjunction or disjunction operators.

$$
\begin{aligned}
Framework &:= \{Clause, \ldots\} \\
Clause &:= Agent :: Dn \\
Agent &:= a(Type, Id) \\
Dn &:= Agent \mid Message \mid Dn\ then\ Dn \mid Dn\ or\ Dn \mid Dn\ par\ Dn \mid null \leftarrow C \\
Message &:= M \Rightarrow Agent \mid M \Rightarrow Agent \leftarrow C \mid M \Leftarrow Agent \mid C \leftarrow M \Leftarrow Agent \\
C &:= Term \mid C \wedge C \mid C \vee C \\
Type &:= Term \\
M &:= Term
\end{aligned}
$$

Where *null* denotes an event which does not involve message passing; $Term$ is a structured term in Prolog syntax and Id is either a variable or a unique identifier for the agent. The operator $\leftarrow$, $\wedge$ and $\vee$ are the normal logical connectives for implication, conjunction and disjunction. $M \Rightarrow A$ denotes that a message, M, is sent out to agent A. $M \Leftarrow A$ denotes that a message, M, from agent A is received. The implication operator dominates the message operators, so for example $M \Rightarrow Agent \leftarrow C$ is scoped as $(M \Rightarrow Agent) \leftarrow C$

Figure 2.1: Syntax of LCC Interaction Framework [21]

A more detailed control structure can also added to the agent definition by using *then*, *par* and *or* operators. The *then* operator enforces a

sequence (temporal dependency) between the definitions, the *par* enables two definitions to be carried out in parallel and the *or* defines the choice.

2.1.3 LCC Interaction Framework Example

We are going to use a simplified version of the LCC example discussed in [22] to describe how LCC interaction protocols can be formulated for multiple agents. This example defines a simple auction scenario where an agent taking the role of auctioneer transitions to the role of a caller of bids for a certain item. Once all the calls for bidding have been made the agent transitions its role once again from caller to the item vendor.

The definition of auctioneer agent is shown below where X is the item to be auctioned to a set of bidders (S) with a reserve price R. The constraint item (X,R) defines the reserve price for the item and the constraint bidder(S) defines the set of bidders.

$$a(auctioneer, A) \ :: \ a(auctioneer(X, \ S, \ R, \ []), A) \ \leftarrow \ item(X, R) \ \wedge \ bidders(S)$$

The auctioneer agent can now be further defined to transition to a caller role and then to a vendor role using the sequence operator *then*. This can be described as shown below:

$$a(auctioneer(X, \ S, \ R, \ Bids), A) \ :: \ a(caller(X, \ S, \ R), A) \\ then \ a(vendor(X, \ S, \ R, \ Bids), A)$$

The caller role for an agent is responsible for sending out bidding invitations to all the registered bidders and this can be represented as shown below:

$$a(caller(X, \ S, \ R), A) \ :: \\ (invite\_bid(X, R) \ \Rightarrow \ a(bidder, B) \ \leftarrow \ S \ = \ [B|Sr] \\ then \ a(caller(X, \ Sr \ , \ R), A) \\ or \ null \ \leftarrow \ S \ = \ []$$

invite_bid message, carrying the item and the reserved price, is sent by the caller to an bidder agent. And this process is recursively repeated for all the registered bidders present in the bidder list.

A vendor agent must collect the bids from all the bidding agents and must send the item sold message to the highest bidder. To keep this example simple we are ignoring other possible outcomes of the auction such as matching the bids to the reserve price and if there is no single highest bidder amongst the collected bids.

$a(vendor(X,\ S,\ R,\ C), A)\ ::$
$add\_bid(Bb,\ Vb,\ C,\ Cn)\ \leftarrow\ bid(X,\ Vb)\ \Leftarrow\ a(bidder, Bb)$
then
$(\ sold(X,\ Vs)\ \Rightarrow\ a(bidder, Bs)\ \leftarrow\ all\_bid(S, Cn)\ \wedge\ highest\_bid(Cn,\ Bs,\ Vs)$
or
$a(vendor(X,\ S,\ R,\ Cn), A)\ \leftarrow\ not(all\ bid(S, Cn))\ )$

As you can see from the definition above that the agent expects to receive a bid from a *bidder* agent and adds it to the list of received bids. It will only send out the item sold message to the highest bidder once all the bids have been gathered otherwise the agent will retain the vendor role and will keep on accepting bids until all bids are in.

The *bidder* agent on the other hand will receive an auction invitation from the auctioneer agent and will send out its bid for the item. It will then expect to receive sold message from the vendor agent or will resume its bidder role. The definition in LCC can be expressed as following:

$a(bidder, B)\ ::\ invite\ bid(X, R)\ \Leftarrow\ a(auctioneer(X,\ ,\ ,\ ), A)\ then$
$bid(X,\ Vb)\ \Rightarrow\ a(vendor(X,\ ,\ ,\ ), A)\ \leftarrow\ bid\ at(X, R,\ Vb)\ then$
$(sold(X,\ Vs)\ \Leftarrow\ a(vendor(X,\ ,\ ,\ ), A)\ or\ a(bidder, B))$

This example shows how using the LCC interaction framework syntax coordination protocols for the multi agents can be specified for an auction scenario. It also shows how constraints over messages sending/receiving can be specified and how an agent can transition from one role to another under protocol specification.
We shall discuss some more LCC protocol examples in the *Algorithm and Evaluation* chapter while explaining how LCC protocols can be built from SCUFL workflows.

2.1.4 LCC Clause Expansion

In order for an agent to conform to a protocol it is necessary for the agent to unpack any protocol that it receives, find out the next protocol permitted step and save the resultant state of the dialog. There can be a number of ways to undertake such unfolding of the protocol but for LCC the most appropriate way was to do the expansion by applying rewrite rules to the dialog state.

Assuming that there is a distributed system infrastructure in place that is able to ensure message communication and delivery to the system, we can now describe a specific format for the message. Each message will consist of (I,M,R,A,P) where I is the unique identifier for the coordination,

M is the message, R is the role expected from the receiving agent, A is the unique identifier for the receiving agent and P is the protocol specification consisting of a set of clauses (C) defining the dialog framework and set of axioms specifying the common knowledge (K).

On receiving a message in the format described above, the agent places the message M_i in its message set for the messages that are currently under consideration. The agent then tries to find a clause C_i that it has indexed under coordination identifier I. If the indexed clause is found it is selected, otherwise the agent makes a copy of the C_i from the set of clauses C. The rewrite rules, discussed in detail in section (4.1), are applied on the C_i considering the protocol (P) and the set of received messages (M_i). This would result in a new dialog clause C_n, an output set of messages O_n and remaining unprocessed set of messages M_n from the original set of messages M_i. Applying the protocol rewrite rules exhaustively would produce the sequence:

$$\left\langle C_i \xrightarrow{M_i, M_{i+1}, \mathcal{P}, O_i} C_{i+1}, C_{i+1} \xrightarrow{M_{i+1}, M_{i+2}, \mathcal{P}, O_{i+1}} C_{i+2}, ..., C_{n-1} \xrightarrow{M_{n-1}, M_n, \mathcal{P}, O_n} C_n \right\rangle$$

The original clause C_i is then replaced in the protocol P by C_n to produce the new protocol, P_n. The copy of new protocol P_n can then be sent along with the output messages set O_n. [22]

2.2 Simple Conceptual Unified Flow Language (SCUFL)

Taverna is an open source workflow system and provides a language and software tools to facilitate easy use of workflow and distributed computer technology within the eScience community. Biologists and bioinformaticians can build highly complex scientific workflows using public and private data with their normal computational resources. These scientists might have a limited knowledge of computational resources but with excellent graphical interfaces provided by Taverna workbench it is straight forward for such users to search and add processing components to their workflows.

Taverna represents the scientific workflow in a workflow language known as SCUFL (simple conceptual unified flow language) and it is primarily meant for the internal use in Taverna project[20]. SCUFL language can be represented in an XML based syntax which we will briefly outline[19].

2.2.1 XSCUFL Syntax

The XML syntax relies on several tags to define a workflow definition. The SCUFL workflow definition will contain an arbitrary number of processor, link, source, sink and coordination elements. The processor elements specify the main atomic processing units used in the workflow, the link elements specify the data flow link between the processors, the source/sink elements specify the workflow inputs/outputs and the coordination elements specify the control links that exist between the processor elements. The general XML structure for the workflow definition can be summarized as follows:

```
<Scufl version(string) log(int)>
-<processor>*
-<link>*
-<source>*
-<sink>*
-<coordination>*
```

We will now take a brief look at the above mentioned elements for the workflow definition.

Processor Element

A Processor element represents the single atomic processing operation that exists in the workflow. The general structure for a processor element is shown below:

```
<processor name(string)>
-<description>?
-<SPEC ELEMENT maxretries(int)? retrydelay(int)? retrybackoff(double)?>
    * <iterationstrategy>?
    * <alternate>*
```

Each processor element can carry a textual description of the processor and will carry a Spec element description of the processor. Spec element specifies the type of processor that is represented by this processor element. There are more than seven types of processors that can be integrated in the workflow through set of plugins that are available in Taverna's enactor. These processor types are string constants, SOAP based web services, local java classes, Soaplab endpoints, Operation on Biomoby service, XSCUFL workflows and Operations based on Talisman scripting engine. Using the spec element attributes it is possible to define the processor's retry behavior.

A workflow designer can also specify any alternate processor that can be called in case this processing element fails. The workflow designer needs to specify the mapping of input and output ports for the alternate processor

with the main processor. Processor element can also be used to define an iteration strategy of the processor's inputs incase a data-type mismatch is detected. We will describe the iteration strategy later in this section.

Link Element

The link element is used to specify the data flow in the workflow. It defines how an output generated by a processor can be associated with an input of the other. For this purpose source and sink attributes are defined that refer to the output port and the input port of the processor respectively. The value of these attributes will be qualified paths of the processor ports they represent.

```
<link source(string) sink(string)>
```

Source and Sink Elements

These Source and Sink elements represent the workflow level inputs and outputs respectively. They can also carry metadata description of the input/output, which can be used to render the output generated to be displayed to the user.

The concept of source/sink element is opposite from the source and sink attributes mentioned in Link element's description. The workflow level input is represented by a source element where as source attribute in the link element represents the output generated by a processor. Similarly the workflow level output is represented by a sink element where as the sink attribute in the link element represents the input to a processor.

Coordination Element

The coordination element specifies the control flow links between the processors. This element has the following structure:

```
<coordination name(string)>
   <condition>
     <state>
     <target>
   <action>
     <target>
     <statechange>
         <from>
         <to>
```

A processor in a SCUFL workflow can have three defined states: Complete, Scheduled and Running. The coordination element can be used to

specify a condition based on the state of a processor (specified through target sub element). This condition when fulfilled enables the action part of the coordination to be carried out. The action part specifies an another processor to transition from one state to another. At the moment the enactor used by Taverna only allows constraints that block a processor from transitioning from scheduled to running until another processor has completed.

Iteration Strategy

As mentioned earlier, an iteration strategy can be defined for a processor element. The iteration in this context would mean repeated application of the processing element over set of multiple input data items. The Taverna framework offers a default iteration strategy if no explicit iteration strategy is defined for a process. If the workflow designer connects an output port of a processor carrying a list of data item to the input port of another processor that only consumes a single data item of the same type, Taverna will detect a data type mismatch error and can then apply an iteration strategy over this processor. All the inputs to the processor in such a case are considered to be lists of data items and a cross product (orthogonal join) of these input data items will be generated and repeated processor calls will be made to generate a list of outputs.

The default iteration strategy offered by Taverna can be overwritten by stating an explicit iteration strategy through iteration strategy element for the processor element.

```
<iterationstrategy>
   <dot> | <cross>
      <dot>* | <cross>* | <iterator name(string)>*
```

The iteration strategy defines two input combining patterns through the dot and the cross operator tags. The dot operator acts on all the input lists simultaneously selecting and iterating over corresponding items at the same time where as the cross operator generates all possible combinations of items in the input lists.

SCUFL Workflow Example

Figure 2.2 shows a simple SCUFL workflow that is provided along with the Taverna workbench. This workflow on execution fetches the daily Dilbert comic from the Dilbert website. The workflow comprises of a “dilbertURL” processor that provides the URL for the website that is to be used for fetching the Dilbet comic. The “getPage” processor is able to fetch the web page that is specified by the “dilbertURL” processor. The web page fetched by the “getPage” processor is then filtered by the “findComicURL” processor

to find the comic image URL by using the regular expression specified by "comicURLRegex" processor. The filtered image URL list that is generated by the "findComicURL" processor is then passed to "getComicStrip" processor which is able to extract the dilbert comic images using the URL list. The extracted images are then passed as the final output for the workflow.

We shall use this workflow as a basic, running example throughout this document, although we have applied our translation to numerous more complex SCUFL workflows. Figure 2.3 shows the SCUFL script for fetch Daily Dilbert comic workflow.

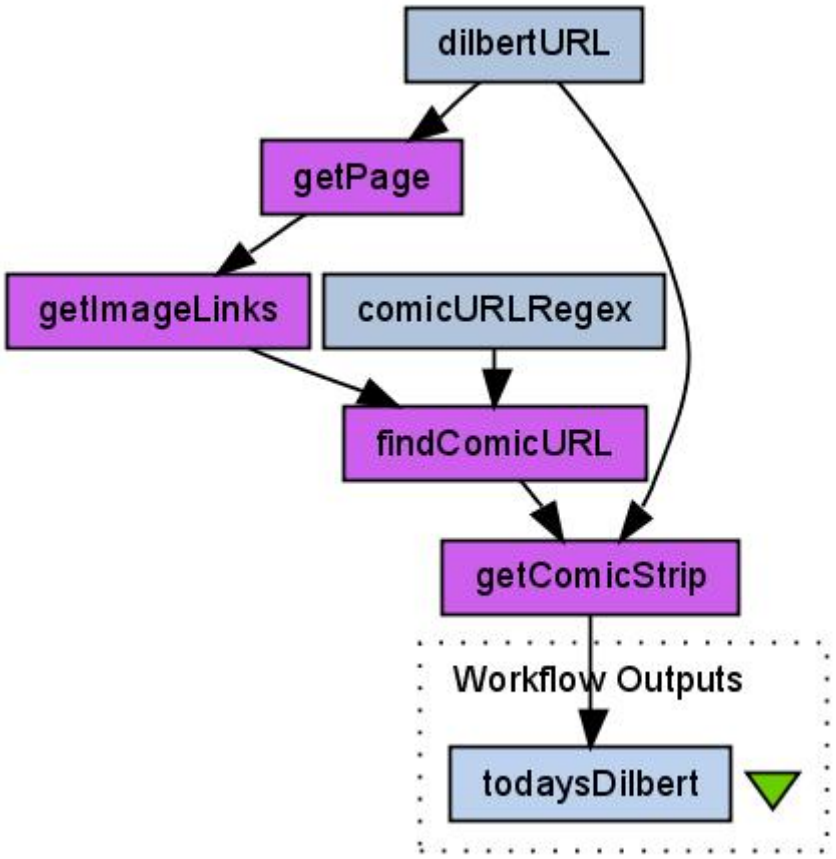

Figure 2.2: An Example SCUFL workflow

```
<?xml version="1.0" encoding="UTF-8"?>
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2" log="0">
  <s:workflowdescription lsid="urn:lsid:www.mygrid.org.uk:operation:VI9FMF5HBQ3"
  author="Tom Oinn" title="Fetch today's Dilbert comic">
  Use the local java plugins and some filtering operations to fetch the comic strip image from
  http://www.dilbert.com
  </s:workflowdescription>
  <s:processor name="dilbertURL">
    <s:stringconstant>http://www.dilbert.com/</s:stringconstant>
  </s:processor>
  <s:processor name="getPage">
    <s:local>org.embl.ebi.escience.scuflworkers.java.WebPageFetcher</s:local>
  </s:processor>
  <s:processor name="getComicStrip">
    <s:local>org.embl.ebi.escience.scuflworkers.java.WebImageFetcher</s:local>
  </s:processor>
  <s:processor name="comicURLRegex">
    <s:stringconstant>.*/archive/images/dilbert.*</s:stringconstant>
  </s:processor>
  <s:processor name="findComicURL">
    <s:local>org.embl.ebi.escience.scuflworkers.java.FilterStringList</s:local>
  </s:processor>
  <s:processor name="getImageLinks">
    <s:local>org.embl.ebi.escience.scuflworkers.java.ExtractImageLinks</s:local>
  </s:processor>
  <s:link source="dilbertURL:value" sink="getPage:url" />
  <s:link source="getPage:contents" sink="getImageLinks:document" />
  <s:link source="getImageLinks:imagelinks" sink="findComicURL:stringlist" />
  <s:link source="comicURLRegex:value" sink="findComicURL:regex" />
  <s:link source="dilbertURL:value" sink="getComicStrip:base" />
  <s:link source="findComicURL:filteredlist" sink="getComicStrip:url" />
  <s:link source="getComicStrip:image" sink="todaysDilbert" />
  <s:sink name="todaysDilbert">
    <s:metadata>
      <s:mimeTypes>
        <s:mimeType>image/*</s:mimeType>
      </s:mimeTypes>
    </s:metadata>
  </s:sink>
</s:scufl>
```

Figure 2.3: SCUFL script for Fetch Daily Dilbert Comic

Chapter 3

Translation Algorithm

We will now take an iterative approach to develop an algorithm that can translate the SCUFL workflow into the LCC protocols for multi-agent workflow system. The first approach that is discussed (Algorithm 1, Section 1.1) focuses on how to produce a basic translation algorithm catering simply to the main gist of the problem.

3.1 Algorithm 1

Before we get into the details of the translation algorithm we will take a brief look at the SCUFL processor in action within the workflow. The example in figure 3.1 shows a specific processor "ShowAnnotatedNonUnique" highlighted from an existing SCUFL workflow. If viewed independently from the rest of the processors, the highlighted processor can also be viewed as an agent whose job is to provide an atomic processing function in the workflow.

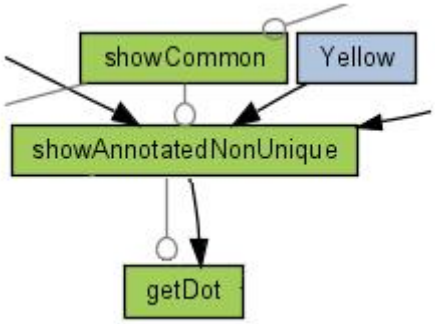

Figure 3.1: A SCUFL Processor in WorkFlow

All the processors, if viewed abstractly, behave in the same algorithmic manner in the workflow i.e. processors expect a certain number of inputs to become available, they can then process the inputs and can then generate an output which can then be sent to a number of down-stream processors. Also, a processor can have associated preconditions that depend on the state of other processors in the workflow and similarly a processor can in turn act as a flow-control condition for other processors. These functions of the processor in the workflow thus give an abstract-template for the processor's behavior and can be directly mapped onto an agent for the processor acting in an open environment.

3.1.1 Introduction to the Agent Protocol's Term and Clauses

Term	Type	Description
doneTransfer	Message	Message sent by an agent to a partner agent downstream to inform that the output has been generated by the agent and is associated with a certain URL carried in the message.
go	Message	Message sent by an agent to another agent who is waiting on the agent for completion of its task. Thus the agent sending the message actually acts as a control precondition for the agent receiving the message.
transfer	Function	This function transforms the received URL for the output produced by the partner agent's processor to the relevant input port of the agent's processor.
perform	Function	This function is associated with the actual execution of the atomic process once all the required preconditions and inputs are available. The perform function can take input arguments and return the output results, depending on the underlying task processing function.

Table 3.1: Defined Terms

While defining an agent for the processor we will have to treat all communication done among the processors as distinct message passing. The data-links (inputs and outputs) and control-links (preconditions and post-conditions) for the processor will now have to be represented through messages. We thus define two simple types of messages to serve this purpose: "go"(for control-links) and "doneTransfer"(for data-links). The "doneTransfer" message will always carry a url:port address within the message content

to specify the location where a certain processor's output is placed. An output URL can then be used by an agent to bind it to a local input port. For this we define a function "transfer" that maps the output port URL onto an input port for an agent's processor. Once an agent receives all the inputs required to proceed, it will then call another function "perform". The "perform" function for an agent will relate to the actual execution of the processor. These defined terms and messages are summarized in table 3.1.

3.1.2 General Definition Template for the Agent

Earlier we mentioned how an agent in the SCUFL workflow appears to follow a certain template of behavior. To summarize any agent in the workflow environment before executing needs to go through the following stages:

1. First, an agent must wait for all the preconditions to become true if there are any.
2. It then has to wait until all the required inputs needed for execution become available from partner processors.
3. Once all the necessary inputs become available for execution the agent would now execute its processor.
4. The result produced can then be sent to down-stream agents through output messages.
5. Once all the output messages are sent out, the agent will then send control-flow messages to the agents that are waiting for this agent to complete its task.

The above mentioned five stages for an agent gives a template of how an agent's behavior for a SCUFL processor needs to generated and this can now be expressed into an LCC protocol script. A thing to note here is that the generated LCC protocol script for an agent might not contain all the behavior stages as shown above and thus the generated script will only contain a certain behavior step if it is found to be present for the processor in the workflow. For example if a processor has no control dependency then the generated LCC protocol script for its agent will not have step 1 & 5 mentioned above.

Using the above template definition we can suggest a general LCC protocol script for any SCUFL processor's agent and this is given below:

```
1-  a(PROCESSOR_NAME, ID) ::
2-  (go() <= a(PROCESSOR_CONTROL, ID))
3-  then
4-  ((transfer(processor_upstream:outport, PROCESSOR_NAME:inputport)) <-
```

```
     (doneTransfer(processor_upstream:outport) <= a(PROCESSOR_UPSTREAM, ID)))
5-  then
6-  ((doneTransfer(PROCESSOR_NAME:outputport) => a(PROCESSOR_DOWNSTREAM, ID))
7-  then
8-  (go() => a(PROCESSOR_WAITING, ID)))
9-       <- perform([PROCESSOR_NAME:inputport], [PROCESSOR_NAME:outputport])
```

The agent definition given above shows the earlier mentioned five stages that an agent might have to go through and also shows how different earlier defined terms and messages are used in the LCC script. If we go through the above definition line by line it is easier to see this correspondence. Line 2 of the script shows how stage 1 of waiting for the precondition to come true is carried out. The agent will have to wait for a *go* message from the agent that holds the control dependency over this agent for execution. Line 4 of the script corresponds to stage 2 of receiving all the necessary input messages (*doneTransfer*) from the partner agents and once the message is received the input port of the agent is bound to the output port contained in the *doneTransfer* message. Line 9 of the script shows that the function *perform* will be called once the precondition and input messages have been received. The perform function will relate to the actual execution of the agent's processing function. The output generated from the processing function will be bound with the output list variable stated in *perform* function. The output messages (*doneTransfer*) and control messages (*go*) will only be sent if the perform function is successful. This is enforced through the implication in line 9. Line 6 shows the output messages being communicated to the agents that are present downstream in the data flow. This corresponds to the earlier described stage 4. The *go* messages shown in line 8 correspond to the earlier described last stage for the agent behavior.

3.1.3 Translation Algorithm

We will now describe how an LCC protocol script for an agent can be generated from the XML based SCUFL workflow definition. The translation algorithm extracts information from the SCUFL XML workflow file and builds the LCC protocol script for each processor's agent. This would thus undertake the agent definition building process similar to the SCUFL agent behavior model described earlier. The algorithm will iterate over existing processors in the SCUFL workflow to produce definitions for their agents. A thing to note here is that the SCUFL workflow has some inputs/outputs that do not belong to any specific processor. So, before any actual translation is carried out, we transform and treat the workflow level input/outputs as belonging to a special agent known as the *System* agent. Thus building the agent definitions for each processor in the workflow we get an additional definition for the *System* agent. This agent deals with workflow level in-

puts/outputs if there are any. The following is a description of how the basic algorithm for translation works:

1. Preprocess the SCUFL file:
   As explained earlier we need to include a *System agent* definition that will incorporate SCUFL workflow inputs/outputs. For this purpose before running the algorithm we need to preprocess the workflow file. During this preprocessing we associate all the ports that are not marked with any processor to the *System* agent. Also note that, as explained earlier, the notion of source/sink port for the workflow is opposite to a source/sink port that belongs to a processor. So, when building the System agent description we use the same algorithm steps as described below but we will interchange the source with sink and vice versa to build a correct agent definition.

2. Get Processor name from the SCUFL workflow:
   This is done by extracting all the "processor" elements that are present in the workflow file. We extract the *name* attribute's value of the processor element and use it as agent's name in the definition. A sample processor element is shown below:

   ```
   <s:processor name="processor_name">...</s:processor>
   ```

   In the generated LCC script for the processor's agent the extracted value is used as shown below:

   ```
   a("processor_name", ID) ::
   ```

3. Generate Preconditions for the Agent definition:
   For the extracted agent name, we now try to extract if the corresponding processor has any preconditions. This can be done by finding coordination elements such that the processor's name exists in the action sub-element and by using the condition sub-element we can locate the agent's name that holds the control flow precondition over this processor's agent. A sample condition element is shown below:

   ```
   <s:coordination name="coordination_name">
     <s:condition>
       <s:state>Completed</s:state>
       <s:target>processor_holding_control</s:target>
     </s:condition>
     <s:action>
       <s:target>processor_name</s:target>
   ```

```
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
```

In the LCC script for the agent definition we will now append the following:

```
a("processor_name", ID) ::
  (go() <= a(processor_holding_control, ID))
```

For every control dependency precondition that is found for this agent an expected go message from the agent holding control condition will be added to the agent's definition separated by a *then* clause.

4. Generate Input messages to be included in the Agent's definition:
We now need to find the agents that send this agent *doneTransfer* messages that will contain the port address that needs to be bound to the relevant input port of the agent's processor. This can be done by searching link elements in the workflow. All the link elements that contain the agent's processor as the input port in the sink attribute can be selected. We can find the corresponding agent responsible for sending the *doneTransfer* message by evaluating the port address that is present in the source attribute of the same link element. If more then one agent is found to be associated with the same input port of the agent, then we need to add an "OR" condition between receiving such messages from the partner agents. The rest of the input message statements are separated by *then* clauses. A sample link element is shown below:

```
<s:link source="proc_up1:value" sink="processor_name:inputport" />
<s:link source="proc_up2:value" sink="processor_name:inputport" />
```

In the LCC script for the agent we will append the following:

```
a("processor_name", ID) ::
      (go() <= a(processor_holding_control, ID))
    then
      ((transfer(proc_up1:value, processor_name:inputport)) <-
        (doneTransfer(proc_up1:value) <= a(proc_up1, ID)))
      or
      ((transfer(proc_up2:value, processor_name:inputport)) <-
        (doneTransfer(proc_up2:value) <= a(proc_up2, ID)))
    then
```

5. Generate Output messages to be sent by the Agent:
   Once we have generated the precondition messages and input messages, we now need to find out the agents that are to be informed of the processing result through an output message from the agent once the processor has done its processing. This can be done again by evaluating the link elements in the workflow. All the link elements that contain the agent's processor name as the output port value in the source attribute can be selected. The corresponding sink attribute value can be used to find the name of the agent that is to be communicated the output port via a *doneTransfer* message. A sample link element is shown below:

   ```
   <s:link source="Echo_list:output" sink="proc_down3:result" />
   ```

   The generated LCC script against such statements is separated by a *then* clause. The LCC script for the above mentioned workflow statement would be following:

   ```
   (doneTransfer(Echo_list:output) => a(proc_down3, ID))
   ```

6. Get Control messages to be sent by the Agent:
   Once the output messages have been generated we now find out if this agent's processor holds a control dependency over some other agent. We again need to evaluate existing coordination elements of the workflow and find out all those coordination elements that contain the agent's processor name in the condition sub-element. We then need to find the agent that is present in the corresponding action sub-element. This action sub-element agent needs to be sent a *go* message. The go messages are appended to the protocol script after all the *doneTransfer* messages have been generated already. A sample coordination element is shown below:

   ```
   <s:coordination name="coordination_name1">
       <s:condition>
         <s:state>Completed</s:state>
         <s:target>processor_name</s:target>
       </s:condition>
       <s:action>
         <s:target>processor_waiting</s:target>
         <s:statechange>
           <s:from>Scheduled</s:from>
           <s:to>Running</s:to>
         </s:statechange>
       </s:action>
     </s:coordination>
   ```

The generated LCC from the above coordination element would look like this:

```
(go() => a(processor_waiting, ID))
```

As there can be more than one coordination condition where the agent acts as a condition, there could be more than one generated LCC statement. In such a scenario these statements are separated from each other using a *then* clause.

7. Add Output Messages and Control Messages to the Agent's Definition: Once all the output messages statements and control message statements are generated, they are put in the same scope using parenthesis and are then made dependent on the *perform* clause using an implication. This will make sure that the agent only sends output and control messages if it is able to perform the processing task successfully. So the above mentioned output and control LCC statements are transformed and appended to the main definition as shown below:

```
a("processor_name", ID) ::
      (go() <= a(processor_holding_control, ID))
    then
      ((transfer(proc_up1:value, processor_name:inputport)) <-
        (doneTransfer(proc_up1:value) <= a(proc_up1, ID)))
      or
      ((transfer(proc_up2:value, processor_name:inputport)) <-
        (doneTransfer(proc_up2:value) <= a(proc_up2, ID)))
    then
      (
        (doneTransfer(processor_name:output) => a(proc_down3, ID))
        then
        (go() => a(processor_waiting, ID))
      ) <- perform([processor_name:inputport], [processor_name:output])
```

3.1.4 Assumptions

We will now highlight some of the underlying assumptions that we made while inventing this algorithm. Firstly, the algorithm does not completely transform the full SCUFL workflow language into the LCC but rather attempts to preserve the basic data and control flow of the workflow. For this basic algorithm we are ignoring the iteration mechanism that is built into SCUFL and is mainly used by Taverna's workflow enactor. The iteration strategy defines how the enactor must process the inputs a processor receives and how it will then generate an output for the processor, but as far as the data flow and control flow of the workflow is concerned it plays

no direct part. We are thus at the moment ignoring iteration strategy in SCUFL workflow.

Secondly, while building the coordination control flow in the LCC we make the same assumption as that of Taverna's workflow enactor. Although syntactically while describing the control flow in SCUFL it is possible to describe a coordination constraint with a condition based on any defined state of one processor which might have an action on another processor to transition its state from any one its defined state to another, Taverna's workflow enactor only allows coordination constraints that block a processor from transitioning from scheduled to running until another processor has achieved a "complete" state. We also make the same assumption about the control flow.

3.1.5 Evaluation

We used the basic algorithm described to build the basic translator module in Java. We are now going to present a simple SCUFL workflow case study and will present the generated LCC script that was obtained from the translator. The first example we are going to look into is a workflow example provided along with Taverna workbench. This workflow compare functions of genes of human Y chromosomes to those on X. This workflow uses BioMart (a query-oriented data management system) over the current Ensembl (a bioinformatics research project) human genome data set to extract all gene ontology terms allocated to genes on X and Y chromosomes. The figure 3.2 shows the workflow layout rendered by Taverna tool.

We are going to list down the LCC script of multiagent interaction definitions required to run this workflow as generated from the Java implementation here. We will only explain one of the generated agent's definition and remaining definitions are only listed afterward.
The LCC agent definition shown in figure 3.3 is for the processor responsible for handling the "getDot" task as shown in figure 3.2. This processor has two control flow dependencies from "showAnnotatedNonUnique" and "showAnnotatedUnique" agents. The "getDot" agent's definition thus specifies that "go" messages should be expected from these agents.

The "getDot" processing task needs the input data to become available from "createSession" agent and for this purpose the definition specifies how this data dependency can be fulfilled using "doneTransfer" message. The output port carried by the "doneTransfer" message is later bound with respective input port of the agent through "transfer" function in the definition. The incoming ports are passed to the processing element of the agent through "perform" function and the generated result is returned to the output argument. If the task processing is done successfuly, this agent must transfer the generated result to the agent downstream in data flow. This is done by sending the "doneTransfer" message carrying the output port to

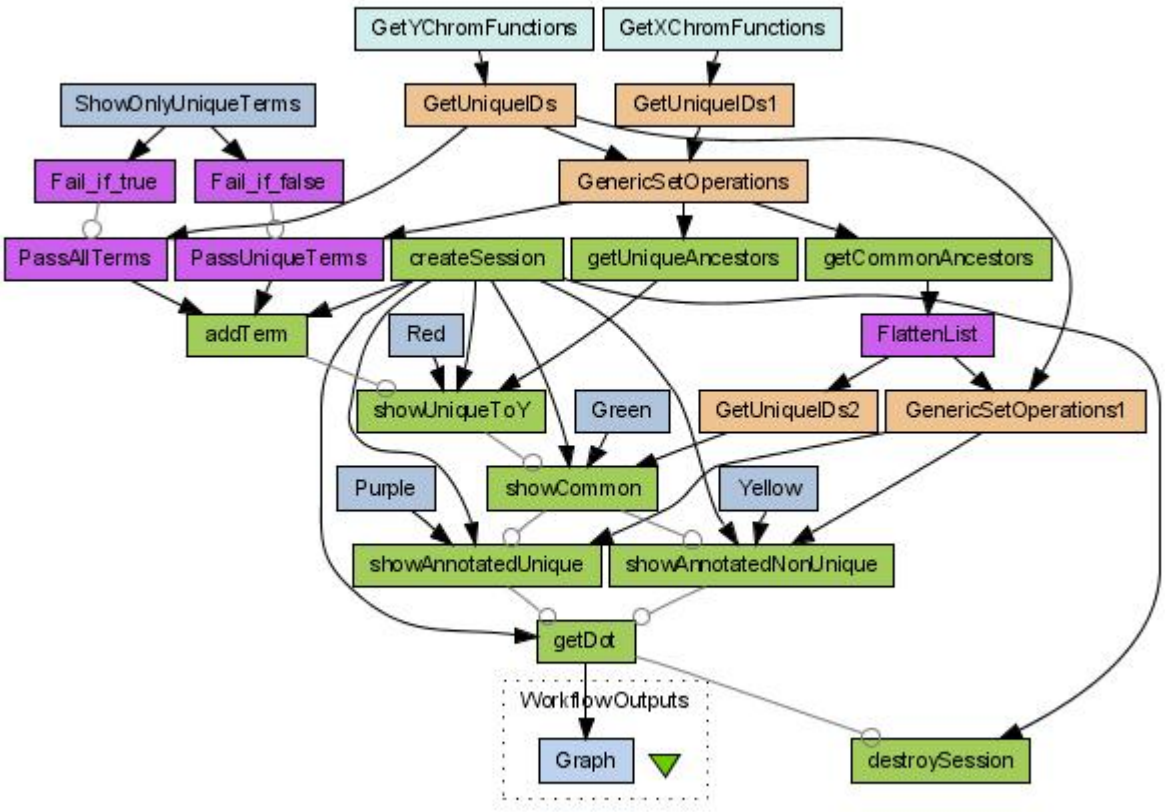

Figure 3.2: SCUFL WorkFlow - Compare X and Y Gene Functions

the concerned agent which is "system" agent in this case. The "getDot" processor holds a control dependency over "destroySession" agent and thus must send a "go" message to this agent once processing is completed successfuly.

```
a(getDot, ID) ::
(go() <= a(showAnnotatedNonUnique, ID))
then
(go() <= a(showAnnotatedUnique, ID))
then
((transfer(createSession:createSessionReturn, getDot:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((doneTransfer(getDot:getDotReturn) => a(system, ID))
then
(go() => a(destroySession, ID))) <- perform([getDot:sessionID], [getDot:getDotReturn])
```

Figure 3.3: LCC Script generated for "getDot" Agent.

The LCC definitions for other agents are given below:

```
a(GenericSetOperations1, ID) ::
((transfer(GetUniqueIDs:out, GenericSetOperations1:set1)) <-
   (doneTransfer(GetUniqueIDs:out) <= a(GetUniqueIDs, ID)))
then
((transfer(FlattenList:outputlist, GenericSetOperations1:set2)) <-
   (doneTransfer(FlattenList:outputlist) <= a(FlattenList, ID)))
then
((doneTransfer(GenericSetOperations1:intersection) => a(showAnnotatedNonUnique, ID))
then
(doneTransfer(GenericSetOperations1:set1not2) => a(showAnnotatedUnique, ID)))
   <- perform([GenericSetOperations1:set1, GenericSetOperations1:set2],
              [GenericSetOperations1:intersection, GenericSetOperations1:set1not2])


a(Red, ID) ::
((doneTransfer(Red:value) => a(showUniqueToY, ID))) <- perform([], [Red:value])


a(GetUniqueIDs, ID) ::
((transfer(GetYChromFunctions:hsapiens_gene_ensembl.go, GetUniqueIDs:in)) <-
   (doneTransfer(GetYChromFunctions:hsapiens_gene_ensembl.go) <= a(GetYChromFunctions, ID)))
then
((doneTransfer(GetUniqueIDs:out) => a(GenericSetOperations1, ID))
then
(doneTransfer(GetUniqueIDs:out) => a(GenericSetOperations, ID))
then
(doneTransfer(GetUniqueIDs:out) => a(PassAllTerms, ID)))
   <- perform([GetUniqueIDs:in], [GetUniqueIDs:out])


a(showCommon, ID) ::
(go() <= a(showUniqueToY, ID))
then
((transfer(createSession:createSessionReturn, showCommon:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((transfer(Green:value, showCommon:colour)) <-
   (doneTransfer(Green:value) <= a(Green, ID)))
then
((transfer(GetUniqueIDs2:out, showCommon:geneOntologyID)) <-
   (doneTransfer(GetUniqueIDs2:out) <= a(GetUniqueIDs2, ID)))
```

```
then
((go() => a(showAnnotatedNonUnique, ID))
then
(go() => a(showAnnotatedUnique, ID)))
   <- perform([showCommon:colour, showCommon:sessionID, showCommon:geneOntologyID], [])


a(PassUniqueTerms, ID) ::
(go() <= a(Fail_if_false, ID))
then
((transfer(GenericSetOperations:set2not1, PassUniqueTerms:inputlist)) <-
   (doneTransfer(GenericSetOperations:set2not1) <= a(GenericSetOperations, ID)))
then
((doneTransfer(PassUniqueTerms:outputlist) => a(addTerm, ID)))
   <- perform([PassUniqueTerms:inputlist], [PassUniqueTerms:outputlist])


a(Fail_if_false, ID) ::
((transfer(ShowOnlyUniqueTerms:value, Fail_if_false:test)) <-
   (doneTransfer(ShowOnlyUniqueTerms:value) <= a(ShowOnlyUniqueTerms, ID)))
then
((go() => a(PassUniqueTerms, ID))) <- perform([Fail_if_false:test], [])


a(GetUniqueIDs1, ID) ::
((transfer(GetXChromFunctions:hsapiens_gene_ensembl.go, GetUniqueIDs1:in)) <-
   (doneTransfer(GetXChromFunctions:hsapiens_gene_ensembl.go) <= a(GetXChromFunctions, ID)))
then
((doneTransfer(GetUniqueIDs1:out) => a(GenericSetOperations, ID)))
   <- perform([GetUniqueIDs1:in], [GetUniqueIDs1:out])


a(FlattenList, ID) ::
((transfer(getCommonAncestors:getAncestorsReturn, FlattenList:inputlist))
   <- (doneTransfer(getCommonAncestors:getAncestorsReturn) <= a(getCommonAncestors, ID)))
then
((doneTransfer(FlattenList:outputlist) => a(GenericSetOperations1, ID))
then
(doneTransfer(FlattenList:outputlist) => a(GetUniqueIDs2, ID)))
   <- perform([FlattenList:inputlist], [FlattenList:outputlist])


a(createSession, ID) ::
((doneTransfer(createSession:createSessionReturn) => a(addTerm, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(destroySession, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(getDot, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(showAnnotatedNonUnique, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(showAnnotatedUnique, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(showCommon, ID))
then
(doneTransfer(createSession:createSessionReturn) => a(showUniqueToY, ID)))
   <- perform([], [createSession:createSessionReturn])


a(getUniqueAncestors, ID) ::
((transfer(GenericSetOperations:set2not1, getUniqueAncestors:geneOntologyID))
   <- (doneTransfer(GenericSetOperations:set2not1) <= a(GenericSetOperations, ID)))
```

```
then
((doneTransfer(getUniqueAncestors:getAncestorsReturn) => a(showUniqueToY, ID)))
   <- perform([getUniqueAncestors:geneOntologyID], [getUniqueAncestors:getAncestorsReturn])


a(GenericSetOperations, ID) ::
((transfer(GetUniqueIDs:out, GenericSetOperations:set2)) <-
   (doneTransfer(GetUniqueIDs:out) <= a(GetUniqueIDs, ID)))
then
((transfer(GetUniqueIDs1:out, GenericSetOperations:set1)) <-
   (doneTransfer(GetUniqueIDs1:out) <= a(GetUniqueIDs1, ID)))
then
((doneTransfer(GenericSetOperations:intersection) => a(getCommonAncestors, ID))
then
(doneTransfer(GenericSetOperations:set2not1) => a(PassUniqueTerms, ID))
then
(doneTransfer(GenericSetOperations:set2not1) => a(getUniqueAncestors, ID)))
   <- perform([GenericSetOperations:set1, GenericSetOperations:set2],
              [GenericSetOperations:intersection, GenericSetOperations:set2not1])


a(getCommonAncestors, ID) ::
((transfer(GenericSetOperations:intersection, getCommonAncestors:geneOntologyID)) <-
   (doneTransfer(GenericSetOperations:intersection) <= a(GenericSetOperations, ID)))
then
((doneTransfer(getCommonAncestors:getAncestorsReturn) => a(FlattenList, ID)))
   <- perform([getCommonAncestors:geneOntologyID], [getCommonAncestors:getAncestorsReturn])


a(destroySession, ID) ::
(go() <= a(getDot, ID))
then
((transfer(createSession:createSessionReturn, destroySession:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
null <- perform([destroySession:sessionID], [])


a(Fail_if_true, ID) ::
((transfer(ShowOnlyUniqueTerms:value, Fail_if_true:test)) <-
   (doneTransfer(ShowOnlyUniqueTerms:value) <= a(ShowOnlyUniqueTerms, ID)))
then
((go() => a(PassAllTerms, ID)))
   <- perform([Fail_if_true:test], [])


a(Purple, ID) ::
((doneTransfer(Purple:value) => a(showAnnotatedUnique, ID)))
   <- perform([], [Purple:value])


a(addTerm, ID) ::
((transfer(createSession:createSessionReturn, addTerm:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((transfer(PassUniqueTerms:outputlist, addTerm:geneOntologyID)) <-
   (doneTransfer(PassUniqueTerms:outputlist) <= a(PassUniqueTerms, ID)))
   or
((transfer(PassAllTerms:outputlist, addTerm:geneOntologyID)) <-
   (doneTransfer(PassAllTerms:outputlist) <= a(PassAllTerms, ID)))
then
((go() => a(showUniqueToY, ID)))
```

```
   <- perform([addTerm:sessionID, addTerm:geneOntologyID], [])


a(PassAllTerms, ID) ::
(go() <= a(Fail_if_true, ID))
then
((transfer(GetUniqueIDs:out, PassAllTerms:inputlist)) <-
   (doneTransfer(GetUniqueIDs:out) <= a(GetUniqueIDs, ID)))
then
((doneTransfer(PassAllTerms:outputlist) => a(addTerm, ID)))
   <- perform([PassAllTerms:inputlist], [PassAllTerms:outputlist])


a(showUniqueToY, ID) ::
(go() <= a(addTerm, ID))
then
((transfer(getUniqueAncestors:getAncestorsReturn, showUniqueToY:geneOntologyID)) <-
   (doneTransfer(getUniqueAncestors:getAncestorsReturn) <= a(getUniqueAncestors, ID)))
then
((transfer(createSession:createSessionReturn, showUniqueToY:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((transfer(Red:value, showUniqueToY:colour)) <- (doneTransfer(Red:value) <= a(Red, ID)))
then
((go() => a(showCommon, ID)))
   <- perform([showUniqueToY:geneOntologyID, showUniqueToY:sessionID, showUniqueToY:colour], [])


a(ShowOnlyUniqueTerms, ID) ::
((doneTransfer(ShowOnlyUniqueTerms:value) => a(Fail_if_false, ID))
then
(doneTransfer(ShowOnlyUniqueTerms:value) => a(Fail_if_true, ID)))
   <- perform([], [ShowOnlyUniqueTerms:value])


a(GetUniqueIDs2, ID) ::
((transfer(FlattenList:outputlist, GetUniqueIDs2:in)) <-
   (doneTransfer(FlattenList:outputlist) <= a(FlattenList, ID)))
then
((doneTransfer(GetUniqueIDs2:out) => a(showCommon, ID)))
   <- perform([GetUniqueIDs2:in], [GetUniqueIDs2:out])


a(showAnnotatedUnique, ID) ::
(go() <= a(showCommon, ID))
then
((transfer(createSession:createSessionReturn, showAnnotatedUnique:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((transfer(Purple:value, showAnnotatedUnique:colour)) <-
   (doneTransfer(Purple:value) <= a(Purple, ID)))
then
((transfer(GenericSetOperations1:set1not2, showAnnotatedUnique:geneOntologyID)) <-
   (doneTransfer(GenericSetOperations1:set1not2) <= a(GenericSetOperations1, ID)))
then
((go() => a(getDot, ID)))
   <- perform([showAnnotatedUnique:sessionID, showAnnotatedUnique:colour,
               showAnnotatedUnique:geneOntologyID], [])


a(Yellow, ID) ::
((doneTransfer(Yellow:value) => a(showAnnotatedNonUnique, ID))) <- perform([], [Yellow:value])
```

```
a(showAnnotatedNonUnique, ID) ::
(go() <= a(showCommon, ID))
then
((transfer(createSession:createSessionReturn, showAnnotatedNonUnique:sessionID)) <-
   (doneTransfer(createSession:createSessionReturn) <= a(createSession, ID)))
then
((transfer(Yellow:value, showAnnotatedNonUnique:colour)) <-
   (doneTransfer(Yellow:value) <= a(Yellow, ID)))
then
((transfer(GenericSetOperations1:intersection, showAnnotatedNonUnique:geneOntologyID)) <-
   (doneTransfer(GenericSetOperations1:intersection) <= a(GenericSetOperations1, ID)))
then
((go() => a(getDot, ID)))
   <- perform([showAnnotatedNonUnique:geneOntologyID, showAnnotatedNonUnique:colour,
               showAnnotatedNonUnique:sessionID], [])


a(Green, ID) ::
((doneTransfer(Green:value) => a(showCommon, ID))) <- perform([], [Green:value])


a(GetXChromFunctions, ID) ::
((doneTransfer(GetXChromFunctions:hsapiens_gene_ensembl.go) => a(GetUniqueIDs1, ID)))
   <- perform([], [GetXChromFunctions:hsapiens_gene_ensembl.go])


a(GetYChromFunctions, ID) ::
((doneTransfer(GetYChromFunctions:hsapiens_gene_ensembl.go) => a(GetUniqueIDs, ID)))
   <- perform([], [GetYChromFunctions:hsapiens_gene_ensembl.go])


a(system, ID) ::
((transfer(getDot:getDotReturn, system:Graph)) <-
   (doneTransfer(getDot:getDotReturn) <= a(getDot, ID)))
```

We also introduced the "Fetch Daily Dilbert Comic" workflow 2.2 in the Background (chapter 2). We use this simple workflow as a running example throughout the rest of our work. The LCC interaction protocol definitions for the multiple agents that are generated for this workflow example are given in figure 3.4 and a version of this is used to demonstrate interaction model execution in figure 4.2.

3.1.6 Analyzing the Generated Agent Protocol Scripts

We will now discuss a simple process that can be used to analyze the generated LCC script for an agent. This process is explained below:

- Select a processor from the workflow (this can be done by just evaluating the workflow image).
- Mark all the processors that have a control-flow edge to this processor. For each marked processor there should be a statement in the agent

```
a(dilbertURL, ID) ::
((doneTransfer(dilbertURL:value) => a(getPage, ID))
then
(doneTransfer(dilbertURL:value) => a(getComicStrip, ID))) <-
   perform([], [dilbertURL:value])


a(getPage, ID) ::
((transfer(dilbertURL:value, getPage:url)) <-
   (doneTransfer(dilbertURL:value) <= a(dilbertURL, ID)))
then
((doneTransfer(getPage:contents) => a(getImageLinks, ID))) <-
   perform([getPage:url], [getPage:contents])


a(getComicStrip, ID) ::
((transfer(findComicURL:filteredlist, getComicStrip:url)) <-
   (doneTransfer(findComicURL:filteredlist) <= a(findComicURL, ID)))
then
((transfer(dilbertURL:value, getComicStrip:base)) <-
   (doneTransfer(dilbertURL:value) <= a(dilbertURL, ID)))
then
((doneTransfer(getComicStrip:image) => a(system, ID))) <-
   perform([getComicStrip:base, getComicStrip:url], [getComicStrip:image])


a(comicURLRegex, ID) ::
((doneTransfer(comicURLRegex:value) => a(findComicURL, ID))) <-
   perform([], [comicURLRegex:value])


a(findComicURL, ID) ::
((transfer(comicURLRegex:value, findComicURL:regex)) <-
   (doneTransfer(comicURLRegex:value) <= a(comicURLRegex, ID)))
then
((transfer(getImageLinks:imagelinks, findComicURL:stringlist)) <-
   (doneTransfer(getImageLinks:imagelinks) <= a(getImageLinks, ID)))
then
((doneTransfer(findComicURL:filteredlist) => a(getComicStrip, ID))) <-
   perform([findComicURL:regex, findComicURL:stringlist], [findComicURL:filteredlist])


a(getImageLinks, ID) ::
((transfer(getPage:contents, getImageLinks:document)) <-
   (doneTransfer(getPage:contents) <= a(getPage, ID)))
then
((doneTransfer(getImageLinks:imagelinks) => a(findComicURL, ID))) <-
   perform([getImageLinks:document], [getImageLinks:imagelinks])


a(system, ID) ::
((transfer(getComicStrip:image, system:todaysDilbert)) <-
   (doneTransfer(getComicStrip:image) <= a(getComicStrip, ID)))
```

Figure 3.4: LCC Interaction model script for multiagent system for "Fetch Daily Dilbert Comic" workflow

definition that specifies an incoming *go* message from the marked processor's agent, where each such statement should be separated through a *then* clause.

- Mark all the processors that have a data-flow edge to this processor. For each marked processor there should be a statement in the agent definition that specifies an incoming *doneTransfer* message from the marked processor's agent. The message argument should be bound with the input port of the processor using the *transfer* function. An implication will bind the two parts of such a statement. If there is more than one statement that refer to the same input port then they should be grouped together using the "OR" clause. The resulting groups of statements should be separated from each other using a *then* clause.

- Mark all the processors that have directed data-flow edge from this processor. For each such marked processor there should be a statement in the definition that specifies an outgoing *doneTransfer* message (carrying the output port url) to the marked processor's agent. All such statements should be separated from each other using a *then* clause.

- Mark all the processors that have directed control-flow edge from this processor. For each such marked processor there should be a statement that specifies an outgoing *go* message to the marked processor's agent.

- The outgoing *doneTransfer* and *go* message statements should be put under the same scope using parenthesis and thus forming a statement block. A *perform* function should have an implication over the statement block. If there are no outgoing output or control message statement then only a *perform* function statement should be printed.

Using the process explained above, the analysis of the generated script reveals that the basic data and control flow depicted by the SCUFL workflow is preserved in the generated LCC protocol script. The same analysis was done by using twenty nine SCUFL workflows that are provided along with Taverna workbench as standard workflow examples. We observed similar results for all these workflow examples. We list three of these Taverna workflows and their generated LCC interaction models in Appendix A.3.

3.2 Algorithm 2

This algorithm is an extended version of the basic algorithm described earlier. We will try to address some of the missing SCUFL bits and some of the assumptions that we made earlier while coming up with algorithm 1. The definitions of terms and process of generating the basic agent definition

script that we described earlier in algorithm 1 remain valid for this extended algorithm.

3.2.1 Handling Iteration Strategy

We have mentioned earlier that Taverna's enactor is able to detect data-type mismatches given a set of inputs. In such a scenario the enactor is then able to enforce an iteration strategy over the inputs. If there is no specific iteration strategy defined for the processor then enactor will carry out the default iteration strategy. The default behavior is to treat each item in the input set as a list of expected input and then to generate processor calls by making all possible combinations of these input list items.

The data-type mismatch detection, which is done by the Taverna's enactor, with reference to the LCC based peer to peer agents is not a straight forward task. This matching problem in context of peer to peer agents is considered to be addressed through Dynamic Ontology Matching. The dynamic ontology matching problem and some of its proposed solutions are discussed by Shvaiko et al[24]. We will not go in detail of the matching problem and will assume that a dynamic ontology matching system is in place in the LCC framework kernel and is able detect a data-type mismatch.

So when an agent detects that an input caused a data-type mismatch, it will try to run the iteration strategy over the inputs. If there is no specific definition for the iteration strategy, the agent runs the default iteration strategy over the inputs. The default iteration strategy generates cross products (orthogonal join) of all the input data items and the processor service call is then made by making all possible combinations of input items. The LCC kernel for an agent can then have a general script that will be able to enforce this default iteration mechanism. A default iteration strategy script is shown as a Prolog definition in figure 3.5.

The default iteration script's main function *Iterate* takes an arbitrary size list of inputs, where each item in the list is a list of expected input. The function generates a set of service calls by generating all possible combinations of call arguments. As all possible combination of arguments are generated, we are thus able to produce the cross product (orthogonal join) of the input arguments as done by Taverna's workflow enactor. Once all the calls and empty outputs are generated, the script will then iterate through the generated calls and the results generated from the actual processing calls are then stored in output list.

Lets consider an example of a SCUFL workflow where the iteration strategy being described will be used. The figure 3.6 shows a part of a simple

```
iterate (Inputs, Outputs, ServiceName)  ←
   setof( c(C,O), service_call(Inputs, O, ServiceName, C), calls)
  ∧  call_all(calls, Outputs).

service_call(Inputs, Outputs, ServiceName, ServiceCall)  ←
   select_inputs(Inputs, Parameters)
    ∧  append( Parameters, [Outputs], Arguments)  ∧
   ServiceCall = .. [ServiceName|Arguments].

select_inputs ([L|T], [X|R])  ←
   member(X,L)  ∧  select_inputs( T,R).
select_inputs ([],[]).

call_all ( [c(C|O)|T], [O|R] )  ←   C  ∧  call_all (T, R).
call_all ([], []).
```

Figure 3.5: Default Iteration Strategy Script

SCUFL workflow. As you can see from the labels in the figure that the "Colours" and "Animals" processors provide a "," separated string to their respective list processors. The list processors take a string as an input and produces a list of strings by splitting the input string using "," as regex. The third processor "ColorAnimals" is a processor that takes two string inputs and produces an output string by concatenating the inputs. An interesting thing to note here is that although the "ColorAnimals" processor can only accept string inputs to call its service but it is actually connected to list processors that produce a list of strings. This is where the workflow enactor would apply the iteration strategy while executing this workflow. The enactor while calling the "ColorAnimals" processor service is able to check the data type mismatch error for the inputs and thus can enforce an iteration strategy. So, if the "ColorList" processor produced an output string list ([red, blue]) and "AnimalList" processor produced an output string list ([Rabbit, Cat]) then enactor on detection of data type mismatch error would enforce a default iteration strategy for calling "ColorAnimals" processor service. The cross product of the two inputs will then be generated and processor service calls will be made on each input combination generated by the product. The "ColorAnimals" processor would thus produce four outputs ([redRabbit, redCat, blueRabbit, blueCat]) using its concatenation operation.

When this workflow is translated into LCC interaction model the agent responsible for "ColorAnimals" processor will have to behave in the same way. The LCC kernel will have to provide the data type mismatch error detection component based on dynamic ontology mapping. The agent will then enforce the default or specified iteration strategy using the iteration components defined in the kernel.

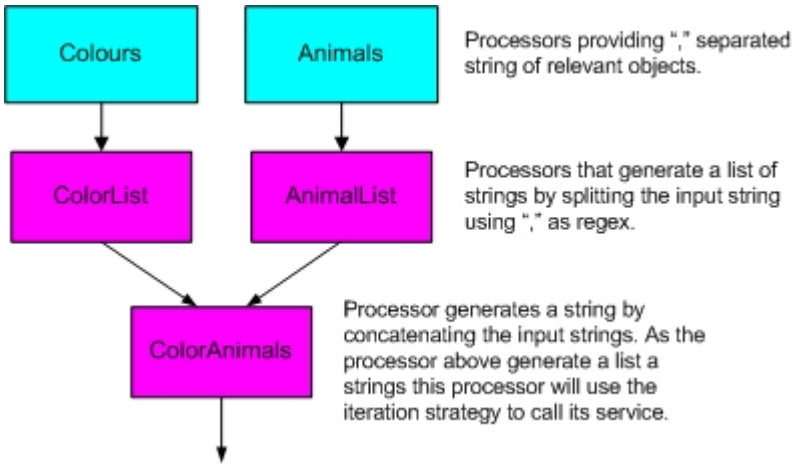

Figure 3.6: Iteration Strategy Example

As mentioned earlier a processor can specify an explicit iteration strategy over its inputs and can thus override the default iteration behavior. This form of iteration strategy can be considered to be a tree of input iterators, where the leaf nodes represent the iteration performed over a single input object and non leaf nodes are combinations of their children based on some defined pattern (dot or cross). Figure 3.7 shows a sample iteration strategy XML block and its conceptual tree representation.

Using a simple tree traversal method, this conceptual tree of iterators can be converted into a simple *iteration strategy* statement that is similar in structure to a prefix notation of the iterator tree. For the iteration strategy example shown in figure 3.7 we can generate a simple *iteration strategy* statement in prefix notation: *"cross([dot([String2, String3]), String1])"*. These generated *iteration strategy* statements can then be passed to a Prolog script component that is given in Appendix A.4. The script is able to apply the *dot* and *cross* operations to the respective input parameters while treating each parameter in the statement as a list. The service calls are generated from the valid parameter item combinations and finally the results obtained

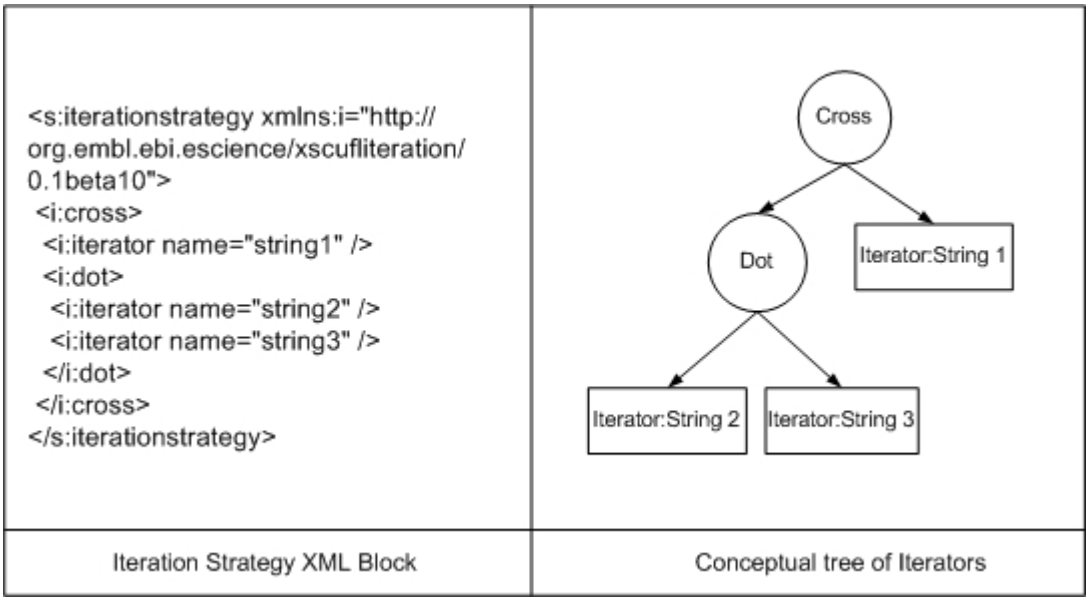

Figure 3.7: Explicit Iteration Strategy for a Processor

from these service calls are returned to the caller.

3.2.2 SCUFL Processor Specific Information

Taverna is able to represent different forms of processing elements in its workflow through SCUFL syntax and there more then seven processing plug-ins that are integrated in Taverna at the moment. These integrated plugins are used to include/run several known forms of services that also include SOAP based WSDL web services. The LCC kernel currently is not able to handle all such processing services although they could be added using the existing facility to share computational components. At the moment we limit ourselves to SOAP based WSDL web service calls. We can use an "invoke" service term in the generated agent protocol script to represent this WSDL web service invocation.

The "perform" function explained earlier in the LCC protocol script generated for a particular agent through algorithm 1 can now be further defined. If the workflow processor refers to SOAP based web service through its WSDL url then the agent definition for this processor can be further elaborated by defining the "perform" function in the following way:

```
perform(Input_arguments, Outputs):-
    invoke_wsdl(wsdl_location, operation_name, Input_arguments, Outputs)
```

In order to provide this extended term definition for "perform" function we need to process the SCUFL workflow file further and extract the required parameters (e.g. WSDL location and operation name for a WSDL processor) from the processor element. In the generated script the parameter shown above will then be replaced by the actual parameter values. The "invoke" directive used in the script will be hard wired in the LCC kernel to provide the necessary processing function.

The iteration strategy definition discussed earlier carries a term "C" which represents a service call being made. This service call can also be interpreted through "perform" function definition present in the extended definition of the agent. In case of a data mismatch error the agent will have to enforce an iteration strategy. The service call ("C") in the strategy definition will now be used in conjunction with the "perform" definition to achieve the necessary processing behavior.

Chapter 4

Simulation Experiment

We now describe the Prolog simulator that we used to test and simulate the LCC interaction models obtained for the SCUFL workflows. The LCC interaction model scripts for the workflow were obtained using the translation algorithm implementation in Java. We describe the operation of the basic Prolog simulator used and also explain our finding through our running example "Fetch Daily Dilbert Comic" SCUFL workflow.

4.1 Prolog Simulator

We used a simple Prolog simulator[1] to run the generated LCC interaction models. The simulator implements the expansion rules described by Robertson, D. in [21]. The expansion rules are stated and explained in the figure 4.1 [21]. The simulator is given a list of existing agents that are expected to be present in the interaction model. For each agent the definition of the agent is extracted from the interaction protocol and the expansion rules are applied to the definition. This expansion process is repeated exhaustively on the body of the definition until no further expansion can be done. The rewrite rules 2-6 in figure 4.1 define how the clause structure of the protocol definition will unfold, that is defined through (*or, then and par*) rewrite rules. The rewrite rules 7-9 explain how a clause term is labeled to be closed once the associated clause constraints terms are justified through either *satisfy or satisfied*. A clause term is considered to be closed completely if all the agent interactions carried by the term are labeled to be closed. A failure to successfully close a certain part of the interaction protocol of an agent's role would result in the simulator highlighting the clause that remained open by printing *"<This section of the clause remains open>"* message. Please refer to the Appendix of the thesis to find the relevant code for the simulator.

[1]The simulator was developed by Robertson, D. as *OpenKnowledge* Prolog interpreter to simulate LCC interaction models.

4.2 Dilbert Experiment

We ran several generated workflow interaction models on the Prolog simulator to evaluate their validity and correctness. Here we are going to describe our finding with respect to the running workflow example of "Fetch Daily Dilbert Comic" given in figure 2.2. The generated LCC interaction protocol script for this workflow is given in figure 3.4. Before running the simulation for this interaction model we assume that the *perform* and *transfer* clauses used in the protocol script will always function right and in this respect will never fail. These were added as "known" facts to the simulation. The figure 4.2 shows the simulation results for running the multiagent interaction model for the "Fetch Daily Dilbert Comic" workflow. The figure 4.2 shows clearly how different agents involved in running the workflow were able to successfully expand their roles using the expansion rules given in figure 4.1.

Notice that the expanded protocol after running appears to be just a sequence of message passing showing which specific interaction messages were passed. If a SCUFL processor in the workflow is unable to perform its task then it can cause the whole workflow to remain incomplete and can cause other depending processors to remain scheduled. Similarly, if an agent is unable to successfully complete its role in the LCC workflow interaction model then it can cause other depending agents also to be unsuccessful in completing their roles. The expansion of an agent role in the interaction model is bounded by how it is able to satisfy constraints. For simulation purposes we assumed *perform* and *transfer* constraints will always be satisfied. But in real deployment it can happen that due to some circumstances an agent might not be able to fulfill the relevant constraint and would thus have same effect on the distributed workflow as a SCUFL processor would have on the SCUFL workflow.

The expanded protocol for an agent can only carry two types of messages: one to cater for the data dependencies (*doneTransfer* message) and the other to cater for the control dependencies (*go* message). It is the correct ordering and sequence of these messages in the expanded protocol that actually is able to ensure the correct working of the workflow protocol. We are also including the expanded clauses in the Appendix for the "Compare X and Y Gene Function" workflow given in figure 3.2. Our experiment for this complicated workflow's LCC interaction model also showed that the simulator was successfully able to close the agents clause terms. We found similar results for other generated workflow interaction model examples.

The following ten rules define a single expansion of a clause. Full expansion of a clause is achieved through exhaustive application of these rules. Rewrite 1 (below) expands a protocol clause with head A and body B by expanding B to give a new body, E. The other nine rewrites concern the operators in the clause body. A choice operator is expanded by expanding either side, provided the other is not already closed (rewrites 2 and 3). A sequence operator is expanded by expanding the first term of the sequence or, if that is closed, expanding the next term (rewrites 4 and 5). A parallel operator expands on both sides (rewrite 6). A message matching an element of the current set of received messages, M_i, expands to a closed message if the constraint, C, attached to that message is satisfied (rewrite 7). A message sent out expands similarly (rewrite 8). A null event can be closed if the constraint associated with it can be satisfied (rewrite 9). An agent role can be expanded by finding a clause in the protocol with a head matching that role and body B - the role being expanded with that body (rewrite 10).

$$A :: B \xrightarrow{M_i,M_o,\mathcal{P},O} A :: E \quad if \quad B \xrightarrow{M_i,M_o,\mathcal{P},O} E$$

$$A_1 \; or \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \quad if \quad \neg closed(A_2) \wedge A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E$$

$$A_1 \; or \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \quad if \quad \neg closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E$$

$$A_1 \; then \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E \; then \; A_2 \quad if \quad A_1 \xrightarrow{M_i,M_o,\mathcal{P},O} E$$

$$A_1 \; then \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} A_1 \; then \; E \quad if \quad closed(A_1) \wedge A_2 \xrightarrow{M_i,M_o,\mathcal{P},O} E$$

$$A_1 \; par \; A_2 \xrightarrow{M_i,M_o,\mathcal{P},O_1 \cup O_2} E_1 \; par \; E_2 \quad if \quad A_1 \xrightarrow{M_i,M_n,\mathcal{P},O_1} E_1 \wedge A_2 \xrightarrow{M_n,M_o,\mathcal{P},O_2} E_2$$

$$C \leftarrow M \Leftarrow A \xrightarrow{M_i,M_i-\{M \Leftarrow A\},\mathcal{P},\emptyset} c(M \Leftarrow A) \quad if \quad (M \Leftarrow A) \in M_i \wedge satisfy(C)$$

$$M \Rightarrow A \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\{M \Rightarrow A\}} c(M \Rightarrow A) \quad if \quad satisfied(C)$$

$$null \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\emptyset} c(null) \quad if \quad satisfied(C)$$

$$a(R,I) \leftarrow C \xrightarrow{M_i,M_o,\mathcal{P},\emptyset} a(R,I) :: B \quad if \quad clause(\mathcal{P}, a(R,I) :: B) \wedge satisfied(C)$$

A protocol term is decided to be closed, meaning that it has been covered by the preceding interaction, as follows:

$$closed(c(X))$$
$$closed(A \; or \; B) \leftarrow closed(A) \vee closed(B)$$
$$closed(A \; then \; B) \leftarrow closed(A) \wedge closed(B)$$
$$closed(A \; par \; B) \leftarrow closed(A) \wedge closed(B)$$
$$closed(X :: D) \leftarrow closed(D)$$

$satisfied(C)$ is true if C can be solved from the agent's current state of knowledge.
$satisfy(C)$ is true if the agent's state of knowledge can be made such that C is satisfied.
$clause(\mathcal{P}, X)$ is true if clause X appears in the dialogue framework of protocol $\mathcal{P}$.

Figure 4.1: Rewrite rules for expansion of a protocol clause [21]

```
a(system,a1)::
    c(doneTransfer(getComicStrip:image)<=a(getComicStrip,a1))

a(getComicStrip,a1)::
    c(doneTransfer(findComicURL:filteredlist)<=a(findComicURL,a1))
    then
    c(doneTransfer(dilbertURL:value)<=a(dilbertURL,a1))
    then
    c(doneTransfer(getComicStrip:image)=>a(system,a1))

a(findComicURL,a1)::
    c(doneTransfer(comicURLRegex:value)<=a(comicURLRegex,a1))
    then
    c(doneTransfer(getImageLinks:imagelinks)<=a(getImageLinks,a1))
    then
    c(doneTransfer(findComicURL:filteredlist)=>a(getComicStrip,a1))

a(getImageLinks,a1)::
    c(doneTransfer(getPage:contents)<=a(getPage,a1))
    then
    c(doneTransfer(getImageLinks:imagelinks)=>a(findComicURL,a1))

a(comicURLRegex,a1)::
    c(doneTransfer(comicURLRegex:value)=>a(findComicURL,a1))

a(getPage,a1)::
    c(doneTransfer(dilbertURL:value)<=a(dilbertURL,a1))
    then
    c(doneTransfer(getPage:contents)=>a(getImageLinks,a1))

a(dilbertURL,a1)::
    c(doneTransfer(dilbertURL:value)=>a(getPage,a1))
    then
    c(doneTransfer(dilbertURL:value)=>a(getComicStrip,a1))
```

Figure 4.2: Fully expanded interaction protocols for "Fetch Daily Dilbert Comic" workflow's multiple agents

Chapter 5

Evaluation

In this chapter we analyze our work so far. We identify the important assumptions while producing the interaction models in LCC from SCUFL. We also address the extent to which these assumptions can be supported by the LCC *OpenKnowledge Kernel*. We will then summarize how much of the SCUFL workflow language can be represented through LCC and vice versa.

5.1 Underlying Assumptions

We described earlier, in the background chapter, retry behavior for a SCUFL processor. The retry behavior is used for a processor if the service it is referring to is found to be out of service or unreachable. The retry behavior specifies the maximum number of retries for the processor's service, the retry delay specifies the delay between each retry attempt and retry backoff defines the backoff time after each retry attempt. This can be done by specifying suitable values for maximum retries, retry delay and retry backoff attributes for the processor. Although it is fairly straight forward to specify such behavior in SCUFL but this behavior cannot be directly specified in the generated LCC protocol. These retry directives in SCUFL are actually used by the underlying workflow enactor which carries out the processor's service call and is thus able to carry out the retry behavior. In LCC we cannot build this retry behavior through the generated interaction model for the agent. The *OpenKnowledge* kernel is still under development and at the moment is not able to provide this functionality. In order to provide the same retry behavior, the *OpenKnowledge* kernel will have to be tweaked to provide the same functionality whenever a service call is made.

We at the moment are ignoring the alternate processor directive that can be specified for a processor in SCUFL. The LCC interaction model we generated are aimed to be run using the *OpenKnowledge* kernel which implicitly has features to find the interaction models that a peer might be interested in. The *OpenKnowledge* system is able to facilitate interaction

model searching and is able to provide interaction models matching to the ones that are required [13]. We thus ignore alternative processor definitions and rely on the kernel to find matching alternative LCC interaction models that will fit the workflow protocol.

We mentioned in the Algorithm chapter that SCUFL language allows several types of processors to be defined but the LCC kernel is currently being developed and is not able to cater to all such processors. At the moment it is only allows WSDL webservice specific calls to be made. Although it is fairly straight forward to translate each SCUFL processor type details to a corresponding LCC clause term by defining *perform* clause further, but to actually run a particular *perform* clause definition would require a specific execution component to be built into the *OpenKnowledge* kernel. For example, one specific processor type in SCUFL is used to provide a string constant. The perform definition for such type of processor's agent is given below. The "STRING_CONSTANT" specified in the definition below in the actual translation will carry the string constant value that is specified by the agent's processor.

```
perform([],[stringConstantAgent:value]) :-
   stringConstantAgent:value = STRING_CONSTANT
```

Another SCUFL processor type is *Local* that is used to define an operation running in the workflow enactor's address space and is based on a local Java class. This operation in LCC can be represented as a constraint term in the perform definition and can be translated by the kernel as a Java operation. We would require a special component to be built in the LCC kernel to handle such Java based constraints.

5.2 Evaluation Summary

We will now summarize our evaluation findings. It is clear from our analysis of the translation that not all SCUFL can be translated into LCC. The basic limiting factor in this respect is the lack of in-built components that are present in the LCC kernel. Although such components can be built in the kernel, at the moment the kernel only supports string constant and WSDL webservice SCUFL processors. The workflow data and control dependencies specified by SCUFL can be fully translated into LCC, which is the main essence of Taverna's graph based workflow. We discussed some translation issues of SCUFL to LCC language in the previous section. We will now summarize all the translation issues while emphasizing the SCUFL constructs and their representation in LCC. The general XML structure for the SCUFL workflow is explained in detail in section 2.2.1 and we will refer to these SCUFL constructs while evaluating. The following is the SCUFL constructs classification that is listed based on the increasing order of difficulty to represent these constructs through LCC:

- The SCUFL constructs that we can translate and execute in LCC at the moment:
  Of the basic SCUFL constructs, the "link" and "coordination" constructs can be fully represented through a corresponding LCC interaction model. These constructs are of great importance as they define the data flow and the control flow for the SCUFL workflow. The generated interaction models in LCC thus also preserve the expected data and control flow. We can also produce a basic level representation for SCUFL "processor" construct. The details of how the translation for these constructs is carried out is defined in the basic translation algorithm section 3.1.3.

- The SCUFL constructs that we could translate and execute with extension to the LCC system:
  A basic representation of SCUFL "processor" construct can be generatd in LCC but specific information contained in "processor" construct may require an extension to the existing LCC *OpenKnowledge* kernel system. We explained in the last section 5.1 that SCUFL allows several types of "processor" to be defined. As far as the translation is concerned any SCUFL processor type can be represented in LCC by a corresponding clause term which will define how the processor execution needs to be carried out by the LCC kernel. But in order to interpret and execute these LCC clauses, specific execution components in LCC kernel will be required. At the moment the LCC kernel is only able to handle two types of SCUFL processors namely "String" type and "WSDL" webservice type processors.

  A SCUFL processor can also specify an explicit iteration strategy which overrides the default iteration strategy. We explained in section 3.2.1 how explicit iteration strategy can be handled in context of LCC. We also presented two Prolog script components that need to be integrated in the LCC kernel in order to carry out the execution of the iteration strategy for a SCUFL processor. Another component emphasized in section 3.2.1 that needs to be developed for the LCC kernel was the "Dynamic Ontology Matching" component to carry out the data type mismatch detection.

  We addressed the above mentioned issues in detail in "Algorithm 2" section 3.2.

- The SCUFL constructs that we do not know precisely how to translate:
  We explained in the last section 5.1 that a SCUFL processor can define "retry behavior" through some "processor" construct attributes. As explained earlier, such retry behavior cannot be directly represented in LCC interaction model. Although in order to provide the same functionality the *OpenKnowledge* kernel for LCC needs to be modified

to provide the same retry effect whenever a service call is made.

We can therefore be confident in saying that LCC language is capable of representing all workflows that can be made using SCUFL. The converse is not true. SCUFL is only able to represent simple graph based workflows but LCC on the other hand can represent even more complicated structured workflows using its logic operators and its ability to use recursive structures. This feature of LCC was highlighted earlier in the LCC interaction framework example discussed in section 2.1.3. Following LCC script taken from the same example emphasizes the recursive nature of LCC language and its ability to unpack a data structures.

$a(caller(X,\ S,\ R), A)\ ::$
$(invite\_bid(X, R)\ \Rightarrow\ a(bidder, B)\ \leftarrow\ S\ =\ [B|Sr]$
$\qquad\qquad\qquad\qquad\qquad\qquad then\ a(caller(X,\ Sr\ ,\ R), A)$
$or\ null\ \leftarrow\ S\ =\ []$

This script shows a caller agent role in an auction scenario that recursively iterates over the bidder list while sending each agent registered in the list an invite bid message carrying the item and its reserved price. Such recursive structures and data structure unpacking feature cannot be produced directly and elegantly through SCUFL and it might require some new service to be invented or some new workflow enactor plug-ins to produce the same effect.

Another important feature of LCC language is that interaction models can be defined in such a way that agents can join an interaction scenario at run time. For example, in an auction scenario a LCC interaction model can be defined such that it can allow bidding agents to join in on an auction at run time. In SCUFL all processors that are part of the workflow are defined before the execution of the workflow. A new processor cannot be added to the workflow while the execution is being carried out. Similarly, through LCC language a peer can ask other peers for a particular interaction model. Required interaction model can then be communicated to the querying peer by a neighboring peer offering the desired interaction model. SCUFL does not offer any such feature.

It is thus fair to conclude that LCC is able to represent the SCUFL language modulo limitations that are mainly associated with the LCC's *Open-Knowledge* kernel but the converse is not true. Considering that these two languages are based on two different paradigms (i.e. the LCC language represents the message passing paradigm where as SCUFL language represents the data flow paradigm), this result conforms to our intuitions that these two language will not be a subset of the other. It is surprising, however, that almost all of the representational ability of SCUFL can be obtained through LCC via a comparatively simple automatic translator.

Chapter 6

Conclusion

Agent and Grid technologies have progressed significantly with time. As mentioned in "Introduction" chapter, while the grid community was primarily focused on developing frameworks to compose and execute workflow, the agent community developed agent based peer-to-peer technologies. There is now an increasing interest in the scientific community to converge these two technologies in order to take advantage of the benefits offered by agent based peer-to-peer technologies. In this context we selected a scientific workflow composition and enactment tool (Taverna) and showed how its underlying workflow definition language (SCUFL) can be mapped and translated into LCC language that is used to define interaction models for multi agent systems.

In "Background" chapter we reviewed in detail both SCUFL and LCC languages, explained in detail the general language structure and syntax. We also explained the main concepts that exist in these languages and emphasized further by presenting relevant examples. In "Translation Algorithm" chapter we addressed the main problem hypothesis that we presented in the "Introduction" chapter. We defined the general protocol terms that are used in the generated LCC interaction model. We then explained how SCUFL structures can be mapped onto corresponding LCC statements and then presented a detailed algorithm for this translation process. We applied this basic translation algorithm on standard SCUFL workflow examples provided along with Taverna workbench and produced corresponding interaction models for multi agents. We extended the basic algorithm further to explain how some missing SCUFL bits and assumptions can be addressed.

We then conducted the simulation experiment on the generated LCC interaction models and presented our findings in the "Simulation Experiment" chapter. The experiments conducted on the generated LCC protocol scripts resulted in successful simulation of the multi agent interaction models and

thus proving the accuracy of our basic translation algorithm. We then evaluated our translation algorithm while emphasizing some of the underlying assumptions and their solutions in the "Evaluation" chapter. We concluded through our evaluation that the LCC language is successfully able to represent the SCUFL language modulo limitations but the converse is not true.

We successfully proved our hypothesis that the LCC language is able to serve the same purpose as the Taverna's underlying workflow language SCUFL and thus it is possible to deploy the Taverna's scientific workflows in a multi agent based system using LCC language. We presented in detail how a general translation algorithm can produce LCC interaction protocols script for multi agents from the SCUFL workflows and also emphasized the extent to which some of the underlying assumptions can be supported by the LCC *OpenKnowledge* kernel. Further components that need to be developed in the kernel can be considered as future work that will eliminate existing limitations highlighted by our work.

Bibliography

[1] Foster, I. and Kesselman, C. eds., *"The Grid: Blueprint for a New Computing Infrastructure"*, Morgan Kaufmann, San Francisco, 1999.

[2] Ludascher, B., Altintas, I., Berkley, C., Higgins, D., Jaeger, E. and Zhao, Y. (2006) *"Scientific workflow management and the Kepler system.* Concurrency And Computation: Practice and Experience, Volume 18:1039-1065.

[3] Majithia, S., Walker, D. W. and Gray, W. A. (2004) *Automating Scientific Experiments on the Semantic Grid.* The Semantic Web ISWC 2004: Third International Semantic Web Conference, Hiroshima, Japan.

[4] Goble, C. and De Roure, D. (2002) *"The Grid: An Application of the Semantic Web"* , www.semanticgrid.org.

[5] Duncan Hull, Katy Wolstencroft, Robert Stevens, Carole Goble, Mathew R. Pocock, Peter Li and Tom Oinn, Taverna: a tool for building and running workflows of services. In Nucleic Acids Research 2006 34(Web Server issue).

[6] *"The Taverna Project"*, http://taverna.sourceforge.net/

[7] *"The Triana Project"*, http://www.trianacode.org/index.html

[8] Wooldridge, M. (1999) *"Intelligent Agents"*, In G. Weiss, editor: Multiagent Systems, The MIT Press.

[9] Jennings, N.R. (2001) *"An agent-based approach for building complex software systems."*, Communications of the ACM, 44 (4). 35-41.

[10] D. Robertson, C. Walton, A. Barker, P. Besana, Y. Chen-Burger, F. Hassan, D. Lambert, G. Li, J .McGinnis, N. Osman, A. Bundy, F. McNeill, F. van Harmelen, C. Sierra, F. Giunchiglia. (2007), *"Models of Interaction as a Grounding for Peer to Peer Knowledge Sharing"*. Advances in Web Semantics.

[11] Barker, A. and Mann, Robert G. (2006) *"Flexible Service Composition"*. In Lecture Notes in Artificial Intelligence, Volume 4149, Springer Verlag.

[12] I. Foster, N. R. Jennings, and C. Kesselman. *"Brain meets Brawn: Why Grid and Agents Need Each Other"*. In Proc. 3rd Int. Conf. on Autonomous Agents and Multi-Agent Systems, New York, USA, 2004.

[13] Robertson, D. (2007) *"OpenKnowledge Manual"*. http://www.cisa.informatics.ed.ac.uk/OK/manual/manual.html.

[14] Smart Grid Technologies Workshop. In Fourth International Joint Conference on Autonomous Agents and Multiagent Systems, Utrecht, The Netherlands, July 2005.

[15] Agent-Based Grid Computing Workshop. In 6th IEEE International Symposium on Cluster Computing and the Grid, Singapore, May 2006.

[16] Dr. Huaglory and Dr. Rainer Unland, editors. *"Multiagent and Grid Systems"*. IOS Press.

[17] Guo, L., Robertson, D., and Chen-Burger, Y. 2005. *"A Novel Approach for Enacting the Distributed Business Workflows Using BPEL4WS on the Multi-Agent Platform"*. In Proceedings of the IEEE international Conference on E-Business Engineering (October 12 - 18, 2005). ICEBE. IEEE Computer Society, Washington, DC, 657-664.

[18] *"Business Process Execution Language For Web Services specification"*, http://www.ibm.com/developerworks/library/specification/ws-bpel//

[19] Oinn, T. (2004) *"XScufl Language Reference"*, http://www.ebi.ac.uk/ tmo/mygrid/XScuflSpecification.html.

[20] University of Southampton, Open Middleware Infrastructure Institute, (2006) *"About Taverna"*.

[21] Robertson, D. (2004) *"Multi-agent Coordination as Distributed Logic Programming"*. In Lecture Notes in Computer Science :416-430, Springer Berlin.

[22] Robertson, D. (2005) *"A Lightweight Coordination Calculus for Agent Systems"*. In Lecture Notes in Computer Science :183-197, Springer Berlin.

[23] Walton, C. and Robertson, D., *"Flexible multi-agent protocols"*. Technical Report EDI-INF-RR-0164, University of Edinburgh, 2002.

[24] Shvaiko, P., Giunchiglia, F., Schorlemmer, M., McNeill, F., Bundy, A., Marchese, M., Yatskevich, M., Zaihrayeu, I., BoHo, Lopez, V., Sabou, M., Abian, J., Siebes, R. and Kotoulas, S., (2006) *"Dynamic ontology matching: A survey"*.

Appendix A

A.1 Simulator Source Code

Following is the source code for the Prolog based simulator used to run the generated LCC interaction protocols for multiagents:

```
% File: simulator.pl
% Author: Dave Robertson
% The simplest simulator for LCC.

:- ensure_loaded(library(system)),
   ensure_loaded(basic),
   ensure_loaded(util),
   ensure_loaded(loader).

% This portray clause is to prevent you seeing too much detail of the
% LCC interaction model when debugging.  Uncomment it in SICStus Prolog
% if you want to see no details of the interaction model when debugging.
% (see SICStus manual for how portray/1 works).

portray(def(_,_,_)) :- print('Definition').

test8 :-
    sim([a(dilbertURL, a1), a(getPage, a1), a(getComicStrip, a1),
         a(comicURLRegex, a1), a(findComicURL, a1), a(getImageLinks, a1),
         a(system, a1)],
        akram1, FProt),
    lcc_display(FProt).

test10 :-
    sim([a(genericSetOperations1, a1),
         a(red, a1),
         a(getUniqueIDs, a1),
         a(showCommon, a1),
```

```
          a(passUniqueTerms, a1),
          a(fail_if_false, a1),
          a(getUniqueIDs1, a1),
          a(flattenList, a1),
          a(createSession, a1),
          a(getUniqueAncestors, a1),
          a(genericSetOperations, a1),
          a(getCommonAncestors, a1),
          a(destroySession, a1),
          a(fail_if_true, a1),
          a(purple, a1),
          a(addTerm, a1),
          a(passAllTerms, a1),
          a(showUniqueToY, a1),
          a(showOnlyUniqueTerms, a1),
          a(getUniqueIDs2, a1),
          a(showAnnotatedUnique, a1),
          a(yellow, a1),
          a(showAnnotatedNonUnique, a1),
          a(green, a1),
          a(getDot, a1),
          a(getXChromFunctions, a1),
          a(getYChromFunctions, a1),
          a(system, a1)],
         akram2, FProt),
    lcc_display(FProt).

s(Roles, File) :-
    sim(Roles, File, FProt),
    lcc_display(FProt).

%***************************************************************************
% The simplest LCC simulator.
% Loads a LCC interaction model from InstitutionFile and simulates a
% run of the model by unfolding its clauses for the given list of Agents.
% Generates different execution paths of the model on backtracking.
% FProt is the unfolded (instantiated) interaction model at the end of
% the simulation (so it contains a description of what happened in each role)

sim(Agents, InstitutionFile, FProt) :-
    load_institution(InstitutionFile, Prot),
    simulate([], Agents, Prot, FProt).

simulate(Ms, Agents, Prot, FProt) :-
```

```
    sim_step(Ms, Agents, Prot, NewMs, EProt), !,
    simulate(NewMs, Agents, EProt, FProt).
simulate(Ms, Agents, Prot, Prot) :-
    \+ sim_step(Ms, Agents, Prot, _, _).

sim_step(Ms, Agents, Prot, NewMs, EProt) :-
    member(Agent, Agents),
    expansion(Agent, Ms, [], Prot, RestMs, OMessages, EProt),
    \+ Prot = EProt,
    append_messages(OMessages, RestMs, NewMs).

append_messages([m(Af,M => At)|T], List, [m(At,M <= Af)|R]) :-
    append_messages(T, List, R).
append_messages([], List, List).

%***********************************************************************
% Pretty printing a set of LCC clauses
% Only here to make the output of the simulator easier on the eye.

lcc_display(def(Clauses,_,_)) :-
    lcc_disp(Clauses).

lcc_disp([]).
lcc_disp([Clause|T]) :-
    lcc_clause_display(Clause, 0), nl,
    lcc_disp(T).

lcc_clause_display(Role ::= Def, Tab) :-
    tab(Tab), write(Role), write('::'), nl,
    Tab1 is Tab + 4,
    lcc_clause_display(Def, Tab1).
lcc_clause_display(A then B, Tab) :-
    lcc_clause_display(A, Tab),
    tab(Tab), write(then), nl,
    lcc_clause_display(B, Tab).
lcc_clause_display(A or B, Tab) :-
    lcc_clause_display(A, Tab),
    tab(Tab), write(or), nl,
    lcc_clause_display(B, Tab).
lcc_clause_display(Term, Tab) :-
    closed(Term),
    tab(Tab), write(Term), nl.
lcc_clause_display(Term, Tab) :-
    \+ closed(Term),
```

```
    tab(Tab), write('<This section of the clause remains open>'), nl.


%***************************************************************************
% File: basic.pl
% Author: Dave Robertson
% The core Prolog mechanism for LCC

:- op(900, xfx, '::='),
   op(900, xfx, '::'),
   op(900, xfx, '>>'),
   op(800, xfx, '=>'),
   op(800, xfx, '<='),
   op(830, xfx, '<--'),
   op(820, xfy, and),
   op(850, xfy, par),
   op(850, xfy, then),
   op(850, xfy, or).

%***************************************************************************
% Basic LCC interpreter

% Exhaustive expansion of an LCC interaction model, P, for a given
% Agent (of the form a(Role,Id)).

expansion(Agent, Ms, Os, P, FinalMs, FinalOs, FinalP) :-
    expansion_step(Agent, Ms, Os, P, NewMs, NewOs, NewP),
    expansion(Agent, NewMs, NewOs ,NewP, FinalMs, FinalOs, FinalP).
expansion(Agent, Ms, Os, P, Ms, Os, P) :-
    \+ expansion_step(Agent, Ms, Os, P, _, _, _).

expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    protocol_select(agent, P, (a(ARole,Id) ::= Def), P1),
    expand_protocol((a(ARole,Id) ::= Def), Role, Id, Ms, Os, P1,
                    NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).
expansion_step(a(Role,Id), Ms, Os, P, NewMs, NewOs, NewP) :-
    \+ protocol_select(agent, P, (a(Role,Id) ::= _), _),
    protocol_member(dialogue, P, Clause),
    Clause = (a(Role,Id) ::= Def),
    expand_protocol((a(Role,Id) ::= Def), Role, Id, Ms, Os, P,
                    NewA, NewMs, NewOs, P2),
    protocol_add(agent, P2, NewA, NewP).
```

```
% One-step expansion of an LCC clause (see LCC papers for explanation)

expand_protocol(Role ::= Def, _, Id, Ms, Os, P, Role ::= E, Mf, Of, Pf) :-
    expand_protocol(Def, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(A or _, Role, Id, Ms, Os, P, E, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(_ or B, Role, Id, Ms, Os, P, E, Mf, Of, Pf) :-
    expand_protocol(B, Role, Id, Ms, Os, P, E, Mf, Of, Pf).
expand_protocol(A then B, Role, Id, Ms, Os, P, EA then B, Mf, Of, Pf) :-
    expand_protocol(A, Role, Id, Ms, Os, P, EA, Mf, Of, Pf).
expand_protocol(A then B, Role, Id, Ms, Os, P, A then EB, Mf, Of, Pf) :-
    closed(A),
    expand_protocol(B, Role, Id, Ms, Os, P, EB, Mf, Of, Pf).
expand_protocol(A par B, Role, Id, Ms, Os, P, EA par EB, Mf, Of, Pf) :-
    expand_protocol(A par B, Role, Id, Ms, Os, P, EA, Mn, On, Pn),
    expand_protocol(A par B, Role, Id, Mn, On, Pn, EB, Mf, Of, Pf).
expand_protocol(C <-- M <= A, Role, Id, Ms, Os, P, c(M <= A), Mf, Os, Pf) :-
    select(m(Role,M <= A), Ms, Mf),
    satisfied(Id, P, C, Pf).
expand_protocol(M => A <-- C, Role, Id, Ms, Os, P, c(M => A), Ms,
                [m(Role,M => A) | Os], Pf) :-
    satisfied(Id, P, C, Pf).
expand_protocol(M <= A, Role, _, Ms, Os, P, c(M <= A), Mf, Os, P) :-
    select(m(Role,M <= A), Ms, Mf).
expand_protocol(M => A, Role, _, Ms, Os, P, c(M => A), Ms,
                [m(Role,M => A) | Os], P).
expand_protocol(Role <-- C, _, Id, Ms, Os, P, Role ::= Def, Ms, Os, Pf) :-
    Role = a(_,_),
    satisfied(Id, P, C, Pf),
    protocol_member(dialogue, P, (Role ::= Def)).
expand_protocol(Role, _, _, Ms, Os, P, Role ::= Def, Ms, Os, P) :-
    Role = a(_,_),
    protocol_member(dialogue, P, (Role ::= Def)).
expand_protocol(null <-- C, _, Id, Ms, Os, P, c(null), Ms, Os, Pf) :-
    satisfied(Id, P, C, Pf).
expand_protocol(null, _, _, Ms, Os, P, c(null), Ms, Os, P).

closed(c(_)).
closed(A or _) :-
    closed(A).
closed(_ or B) :-
    closed(B).
closed(A then B) :-
```

```
    closed(A),
    closed(B).
closed(A par B) :-
    closed(A),
    closed(B).
closed(_ ::= Def) :-
    closed(Def).

%************************************************************************
% LCC-specific utilities for accessing the data structure that describes
% the interaction model.  The reason for these is to make the definition
% of LCC clause expansion (above) as abstract as possible.

protocol_component(agents, def(Clauses, _, _), Clauses).
protocol_component(dialogue, def(_, Clauses, _), Clauses).
protocol_component(common_knowledge, def(_, _, Clauses), Clauses).

protocol_member(agent, def(Clauses,_,_), Clause) :-
    member(Clause, Clauses).
protocol_member(dialogue, def(_,Clauses,_), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).
protocol_member(common_knowledge, def(_,_,Clauses), ClauseCopy) :-
    member(Clause, Clauses),
    copy_term(Clause, ClauseCopy).

protocol_select(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
protocol_select(dialogue, def(A,Clauses,B), ClauseCopy, def(A,R,B)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).
protocol_select(common_knowledge, def(A,B,Clauses), ClauseCopy, def(A,B,R)) :-
    select(Clause, Clauses, R),
    copy_term(Clause, ClauseCopy).

protocol_remove(agent, def(Clauses,A,B), Clause, def(R,A,B)) :-
    select(Clause, Clauses, R).
protocol_remove(dialogue, def(A,Clauses,B), Clause, def(A,R,B)) :-
    select(Clause, Clauses, R).
protocol_remove(common_knowledge, def(A,B,Clauses), Clause, def(A,B,R)) :-
    select(Clause, Clauses, R).

protocol_add(agent, def(Clauses,A,B), X, def([X|Clauses],A,B)).
protocol_add(dialogue, def(A,Clauses,B), X, def(A,[X|Clauses],B)).
```

```
protocol_add(common_knowledge, def(A,B,Clauses), X, def(A,B,[X|Clauses])).

%*********************************************************************
% Constraint satisfaction code.
% This is just a standard sort of meta-interpreter for conjunctive
% constraints, with the ability to look in the common knowledge
% section of the interaction model for (Horn) clauses.
% Many other forms of constraint handler have been built.

satisfied(Id, P, A and B, Pf) :- !,
    satisfied(Id,  P, A, Pn),
    satisfied(Id,  Pn, B, Pf).
satisfied(Id, P, X, Pf) :-
    meta_pred(Id, X, P, Pf, Call), !,
    Call.
satisfied(Id, P, X, P) :-
    \+ meta_pred(Id, X, P, _, _),
    call_direct(X),
    X.
satisfied(Id,  P, X, P) :-
    protocol_member(common_knowledge, P, known(Id, X)).
satisfied(Id, P, X, Pf) :-
    protocol_member(common_knowledge, P, known(Id, X <-- C)),
    satisfied(Id, P, C, Pf).

call_direct(X) :-
    (predicate_property(X, built_in) ;
     predicate_property(X, interpreted) ;
     predicate_property(X, imported_from(_))), !.

meta_pred(Id, not(X), P, P, \+ satisfied(Id,P,X,_)).
meta_pred(Id, retract(X), P, Pf,
          protocol_remove(common_knowledge,P,known(Id,X),Pf)).
meta_pred(Id, assert(X), P, Pf,
          protocol_add(common_knowledge,P,known(Id,X),Pf)).
```

A.2 Simulation Results for *"Compare X and Y Gene Functions"* Workflow

The following is the expanded clause terms for the generated LCC interaction model for *"Compare X and Y Gene Functions"* workflow example (Figure 3.2):

```
a(system,a1)::
    c(doneTransfer(getDot:getDotReturn)<=a(getDot,a1))

a(destroySession,a1)::
    c(go<=a(getDot,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(null)

a(getDot,a1)::
    c(go<=a(showAnnotatedNonUnique,a1))
    then
    c(go<=a(showAnnotatedUnique,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(getDot:getDotReturn)=>a(system,a1))
    then
    c(go=>a(destroySession,a1))

a(showAnnotatedNonUnique,a1)::
    c(go<=a(showCommon,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(yellow:value)<=a(yellow,a1))
    then
    c(doneTransfer(genericSetOperations1:intersection)<=a(genericSetOperations1,a1))
    then
    c(go=>a(getDot,a1))

a(showAnnotatedUnique,a1)::
    c(go<=a(showCommon,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(purple:value)<=a(purple,a1))
    then
    c(doneTransfer(genericSetOperations1:set1not2)<=a(genericSetOperations1,a1))
    then
    c(go=>a(getDot,a1))

a(showCommon,a1)::
    c(go<=a(showUniqueToY,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(green:value)<=a(green,a1))
    then
    c(doneTransfer(getUniqueIDs2:out)<=a(getUniqueIDs2,a1))
    then
    c(go=>a(showAnnotatedNonUnique,a1))
    then
    c(go=>a(showAnnotatedUnique,a1))

a(getUniqueIDs2,a1)::
    c(doneTransfer(flattenList:outputlist)<=a(flattenList,a1))
    then
    c(doneTransfer(getUniqueIDs2:out)=>a(showCommon,a1))

a(showUniqueToY,a1)::
```

```
    c(go<=a(addTerm,a1))
    then
    c(doneTransfer(getUniqueAncestors:getAncestorsReturn)<=a(getUniqueAncestors,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(red:value)<=a(red,a1))
    then
    c(go=>a(showCommon,a1))

a(genericSetOperations1,a1)::
    c(doneTransfer(getUniqueIDs:out)<=a(getUniqueIDs,a1))
    then
    c(doneTransfer(flattenList:outputlist)<=a(flattenList,a1))
    then
    c(doneTransfer(genericSetOperations1:intersection)=>a(showAnnotatedNonUnique,a1))
    then
    c(doneTransfer(genericSetOperations1:set1not2)=>a(showAnnotatedUnique,a1))

a(flattenList,a1)::
    c(doneTransfer(getCommonAncestors:getAncestorsReturn)<=a(getCommonAncestors,a1))
    then
    c(doneTransfer(flattenList:outputlist)=>a(genericSetOperations1,a1))
    then
    c(doneTransfer(flattenList:outputlist)=>a(getUniqueIDs2,a1))

a(getCommonAncestors,a1)::
    c(doneTransfer(genericSetOperations:intersection)<=a(genericSetOperations,a1))
    then
    c(doneTransfer(getCommonAncestors:getAncestorsReturn)=>a(flattenList,a1))

a(getUniqueAncestors,a1)::
    c(doneTransfer(genericSetOperations:set2not1)<=a(genericSetOperations,a1))
    then
    c(doneTransfer(getUniqueAncestors:getAncestorsReturn)=>a(showUniqueToY,a1))

a(passUniqueTerms,a1)::
    c(go<=a(fail_if_false,a1))
    then
    c(doneTransfer(genericSetOperations:set2not1)<=a(genericSetOperations,a1))
    then
    c(doneTransfer(passUniqueTerms:outputlist)=>a(addTerm,a1))

a(genericSetOperations,a1)::
    c(doneTransfer(getUniqueIDs:out)<=a(getUniqueIDs,a1))
    then
    c(doneTransfer(getUniqueIDs1:out)<=a(getUniqueIDs1,a1))
    then
    c(doneTransfer(genericSetOperations:intersection)=>a(getCommonAncestors,a1))
    then
    c(doneTransfer(genericSetOperations:set2not1)=>a(passUniqueTerms,a1))
    then
    c(doneTransfer(genericSetOperations:set2not1)=>a(getUniqueAncestors,a1))

a(addTerm,a1)::
    c(doneTransfer(createSession:createSessionReturn)<=a(createSession,a1))
    then
    c(doneTransfer(passAllTerms:outputlist)<=a(passAllTerms,a1))
    then
    c(go=>a(showUniqueToY,a1))

a(passAllTerms,a1)::
```

```
    c(go<=a(fail_if_true,a1))
    then
    c(doneTransfer(getUniqueIDs:out)<=a(getUniqueIDs,a1))
    then
    c(doneTransfer(passAllTerms:outputlist)=>a(addTerm,a1))

a(getUniqueIDs,a1)::
    c(doneTransfer(getYChromFunctions:hsapiens_gene_ensembl_go)<=a(getYChromFunctions,a1))
    then
    c(doneTransfer(getUniqueIDs:out)=>a(genericSetOperations1,a1))
    then
    c(doneTransfer(getUniqueIDs:out)=>a(genericSetOperations,a1))
    then
    c(doneTransfer(getUniqueIDs:out)=>a(passAllTerms,a1))

a(getYChromFunctions,a1)::
    c(doneTransfer(getYChromFunctions:hsapiens_gene_ensembl_go)=>a(getUniqueIDs,a1))

a(getUniqueIDs1,a1)::
    c(doneTransfer(getXChromFunctions:hsapiens_gene_ensembl_go)<=a(getXChromFunctions,a1))
    then
    c(doneTransfer(getUniqueIDs1:out)=>a(genericSetOperations,a1))

a(getXChromFunctions,a1)::
    c(doneTransfer(getXChromFunctions:hsapiens_gene_ensembl_go)=>a(getUniqueIDs1,a1))

a(green,a1)::
    c(doneTransfer(green:value)=>a(showCommon,a1))

a(yellow,a1)::
    c(doneTransfer(yellow:value)=>a(showAnnotatedNonUnique,a1))

a(fail_if_true,a1)::
    c(doneTransfer(showOnlyUniqueTerms:value)<=a(showOnlyUniqueTerms,a1))
    then
    c(go=>a(passAllTerms,a1))

a(fail_if_false,a1)::
    c(doneTransfer(showOnlyUniqueTerms:value)<=a(showOnlyUniqueTerms,a1))
    then
    c(go=>a(passUniqueTerms,a1))

a(showOnlyUniqueTerms,a1)::
    c(doneTransfer(showOnlyUniqueTerms:value)=>a(fail_if_false,a1))
    then
    c(doneTransfer(showOnlyUniqueTerms:value)=>a(fail_if_true,a1))

a(purple,a1)::
    c(doneTransfer(purple:value)=>a(showAnnotatedUnique,a1))

a(createSession,a1)::
    c(doneTransfer(createSession:createSessionReturn)=>a(addTerm,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(destroySession,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(getDot,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(showAnnotatedNonUnique,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(showAnnotatedUnique,a1))
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(showCommon,a1))
```

```
    then
    c(doneTransfer(createSession:createSessionReturn)=>a(showUniqueToY,a1))

a(red,a1)::
    c(doneTransfer(red:value)=>a(showUniqueToY,a1))
```

A.3 Taverna Workflow Examples

We applied our translation algorithm on all the Taverna workflow examples. We are only listing a few of these workflow examples due to space constraints.

A.3.1 Fetch Dragon images from BioMoby Example:

This Taverna workflow example shows how Dragon flower images can be extracted using the BioMoby webservices. Figure A.1 shows workflow graph as rendered by the Taverna.

The SCUFL script for "Fetch Dragon Images" workflow is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2" log="0">
  <s:workflowdescription lsid="urn:lsid:www.mygrid.org.uk:operation:TPZT6DLP5V39"
  author="Tom Oinn" title="Fetch Dragon images from BioMoby">Use the local java plugins
  and some filtering operations to fetch the images </s:workflowdescription>
  <s:processor name="id" boring="true">
    <s:stringconstant>cho</s:stringconstant>
  </s:processor>
  <s:processor name="namespace" boring="true">
    <s:stringconstant>DragonDB:Allele</s:stringconstant>
  </s:processor>
  <s:processor name="Decode_base64_to_byte">
    <s:local>org.embl.ebi.escience.scuflworkers.java.DecodeBase64</s:local>
  </s:processor>
  <s:processor name="Parse_moby_data___updated__2006">
    <s:local>org.biomoby.client.taverna.plugin.ExtractMobyData</s:local>
  </s:processor>
  <s:processor name="getJpegFromAnnotatedImage">
    <s:description>This service takes in an annotated jpeg image and
     returns just the image.</s:description>
    <s:biomobywsdl>
      <s:mobyEndpoint>http://mobycentral.icapture.ubc.ca/cgi-bin/MOBY05/mobycentral.pl
      </s:mobyEndpoint>
      <s:serviceName>getJpegFromAnnotatedImage</s:serviceName>
      <s:authorityName>bioinfo.icapture.ubc.ca</s:authorityName>
    </s:biomobywsdl>
  </s:processor>
  <s:processor name="getDragonSimpleAnnotatedImages">
    <s:description>Consumes DragonDB allele identifiers and returns
    a collection of images of plants homozygous for that allele</s:description>
    <s:biomobywsdl>
      <s:mobyEndpoint>http://mobycentral.icapture.ubc.ca/cgi-bin/MOBY05/mobycentral.pl</s:mobyEndpoint>
      <s:serviceName>getDragonSimpleAnnotatedImages</s:serviceName>
      <s:authorityName>antirrhinum.net</s:authorityName>
    </s:biomobywsdl>
  </s:processor>
  <s:processor name="Object">
    <s:description>an object</s:description>
```

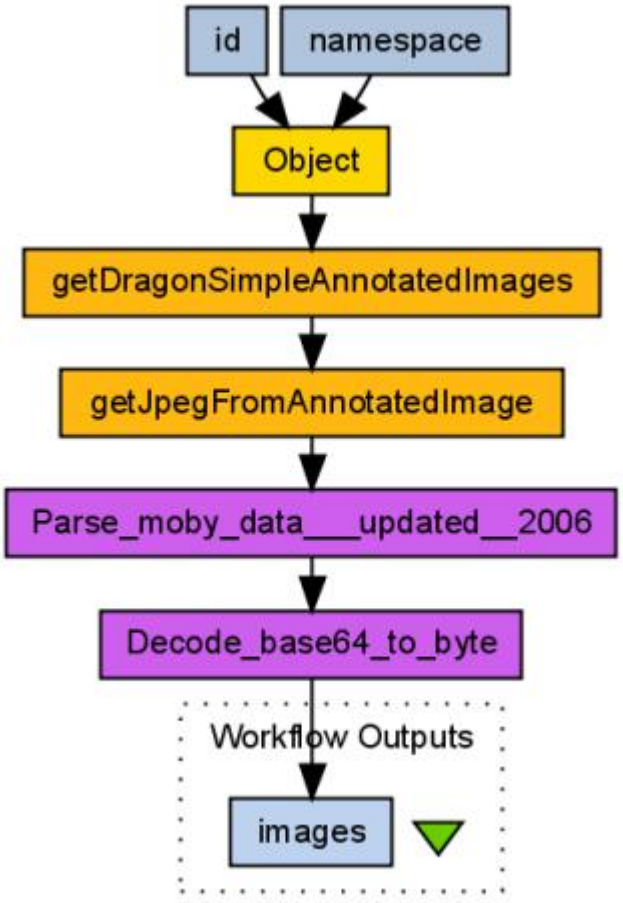

Figure A.1: Taverna Workflow to fetch Dragon images from BioMoby

```
    <s:biomobyobject>
      <s:mobyEndpoint>http://mobycentral.icapture.ubc.ca/cgi-bin/MOBY05/mobycentral.pl
      </s:mobyEndpoint>
      <s:serviceName>Object</s:serviceName>
      <s:authorityName />
    </s:biomobyobject>
  </s:processor>
  <s:link source="Decode_base64_to_byte:bytes" sink="images" />
  <s:link source="Object:mobyData" sink="getDragonSimpleAnnotatedImages:Object(input)" />
  <s:link source="Parse_moby_data___updated__2006:value" sink="Decode_base64_to_byte:base64" />
  <s:link source="getDragonSimpleAnnotatedImages:SimpleAnnotatedJPEGImage(Collection - 'image' As Simples)"
   sink="getJpegFromAnnotatedImage:SimpleAnnotatedJPEGImage(annotatedImage)" />
  <s:link source="getJpegFromAnnotatedImage:b64_encoded_jpeg(image)"
   sink="Parse_moby_data___updated__2006:mobydata" />
  <s:link source="id:value" sink="Object:id" />
  <s:link source="namespace:value" sink="Object:namespace" />
  <s:sink name="images">
    <s:metadata>
      <s:mimeTypes>
        <s:mimeType>image/*</s:mimeType>
      </s:mimeTypes>
    </s:metadata>
  </s:sink>
</s:scufl>
```

LCC interaction models for multiple agents as generated from translation algorithm for the "Fetch Dragon Images" workflow are given below:

```
a(id, ID) ::
((doneTransfer(id:value) => a(Object, ID))) <- perform([], [id:value])

a(namespace, ID) ::
((doneTransfer(namespace:value) => a(Object, ID))) <- perform([], [namespace:value])

a(Decode_base64_to_byte, ID) ::
((transfer(Parse_moby_data___updated__2006:value, Decode_base64_to_byte:base64)) <-
   (doneTransfer(Parse_moby_data___updated__2006:value) <= a(Parse_moby_data___updated__2006, ID)))
then
((doneTransfer(Decode_base64_to_byte:bytes) => a(system, ID))) <-
   perform([Decode_base64_to_byte:base64], [Decode_base64_to_byte:bytes])

a(Parse_moby_data___updated__2006, ID) ::
((transfer(getJpegFromAnnotatedImage:b64_encoded_jpeg(image), Parse_moby_data___updated__2006:mobydata)) <-
   (doneTransfer(getJpegFromAnnotatedImage:b64_encoded_jpeg(image)) <= a(getJpegFromAnnotatedImage, ID)))
then
((doneTransfer(Parse_moby_data___updated__2006:value) => a(Decode_base64_to_byte, ID))) <-
   perform([Parse_moby_data___updated__2006:mobydata], [Parse_moby_data___updated__2006:value])

a(getJpegFromAnnotatedImage, ID) ::
((transfer(getDragonSimpleAnnotatedImages:SimpleAnnotatedJPEGImage(Collection - 'image' As Simples),
     getJpegFromAnnotatedImage:SimpleAnnotatedJPEGImage(annotatedImage))) <-
   (doneTransfer(getDragonSimpleAnnotatedImages:SimpleAnnotatedJPEGImage(Collection - 'image' As Simples))
     <= a(getDragonSimpleAnnotatedImages, ID)))
then
((doneTransfer(getJpegFromAnnotatedImage:b64_encoded_jpeg(image)) => a(Parse_moby_data___updated__2006, ID))) <-
   perform([getJpegFromAnnotatedImage:SimpleAnnotatedJPEGImage(annotatedImage)],
    [getJpegFromAnnotatedImage:b64_encoded_jpeg(image)])

a(getDragonSimpleAnnotatedImages, ID) ::
((transfer(Object:mobyData, getDragonSimpleAnnotatedImages:Object(input))) <-
   (doneTransfer(Object:mobyData) <= a(Object, ID)))
then
```

```
((doneTransfer(getDragonSimpleAnnotatedImages:SimpleAnnotatedJPEGImage(Collection - 'image' As Simples))
  => a(getJpegFromAnnotatedImage, ID))) <-
   perform([getDragonSimpleAnnotatedImages:Object(input)],
           [getDragonSimpleAnnotatedImages:SimpleAnnotatedJPEGImage(Collection - 'image' As Simples)])

a(Object, ID) ::
((transfer(namespace:value, Object:namespace)) <- (doneTransfer(namespace:value) <= a(namespace, ID)))
then
((transfer(id:value, Object:id)) <- (doneTransfer(id:value) <= a(id, ID)))
then
((doneTransfer(Object:mobyData) => a(getDragonSimpleAnnotatedImages, ID))) <-
   perform([Object:id, Object:namespace], [Object:mobyData])

a(system, ID) ::
((transfer(Decode_base64_to_byte:bytes, system:images)) <-
   (doneTransfer(Decode_base64_to_byte:bytes) <= a(Decode_base64_to_byte, ID)))
```

A.3.2 Show Gene Ontology Context Workflow Example:

This Taverna workflow example builds up a subgraph of the Gene Ontology (http://www.geneontology.org) to show the context for a supplied term or terms. Figure A.2 shows workflow graph as rendered by the Taverna.

The SCUFL script for "Show Gene Ontology Context" workflow is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2" log="0">
  <s:workflowdescription lsid="urn:lsid:www.mygrid.org.uk:operation:M763988GGM1" author="Tom Oinn"
  title="Show Gene Ontology Term Context">This workflow builds up a subgraph of the Gene Ontology
  (http://www.geneontology.org) to show the context for a supplied term or terms. It shows this context
  by colouring all ancestors of the term, all children and all siblings. By default, ancestors of the
  supplied term or terms are coloured orange, siblings purple and direct children teal. Other terms
  appear in the default wheat colour.</s:workflowdescription>
  <s:processor name="ancestorColour" boring="true">
    <s:stringconstant>gold1</s:stringconstant>
  </s:processor>
  <s:processor name="getChildren">
    <s:arbitrarywsdl>
      <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
      <s:operation>getChildren</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="getAncestry">
    <s:arbitrarywsdl>
      <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
      <s:operation>getAncestors</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="getParents">
    <s:arbitrarywsdl>
      <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
      <s:operation>getParents</s:operation>
    </s:arbitrarywsdl>
  </s:processor>
  <s:processor name="inputTermColour" boring="true">
    <s:stringconstant>darkolivegreen3</s:stringconstant>
  </s:processor>
  <s:processor name="colourInputTerm">
```

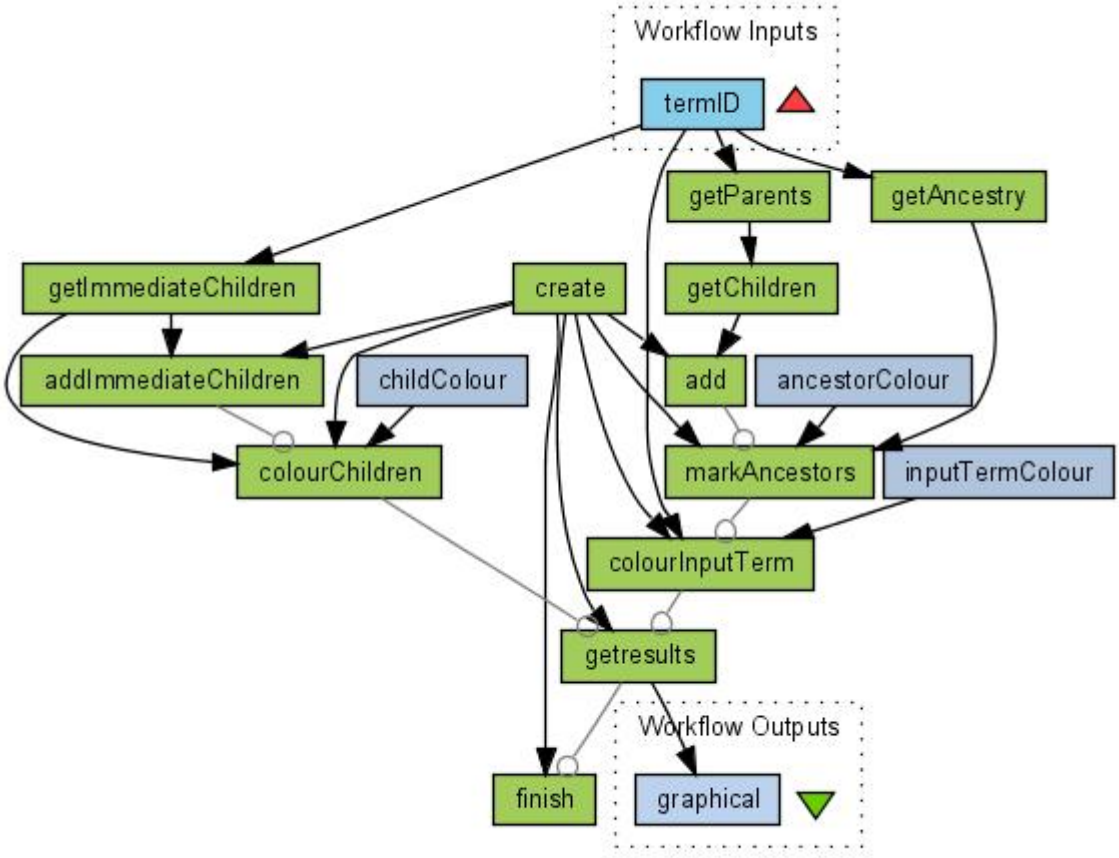

Figure A.2: Taverna Workflow to show Gene Ontology Context for supplied terms

```
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>markTerm</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="getImmediateChildren">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>getChildren</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="markAncestors">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>markTerm</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="getresults">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>getDot</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="finish">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>destroySession</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="add">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>addTerm</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="addImmediateChildren">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>addTerm</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="childColour" boring="true">
  <s:stringconstant>darkslategray3</s:stringconstant>
</s:processor>
<s:processor name="colourChildren">
  <s:arbitrarywsdl>
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>markTerm</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:processor name="create">
  <s:arbitrarywsdl critical="true">
    <s:wsdl>http://www.ebi.ac.uk/collab/mygrid/service1/goviz/GoViz.jws?wsdl</s:wsdl>
    <s:operation>createSession</s:operation>
  </s:arbitrarywsdl>
</s:processor>
<s:link source="termID" sink="getParents:geneOntologyID" />
<s:link source="ancestorColour:value" sink="markAncestors:colour" />
<s:link source="create:createSessionReturn" sink="add:sessionID" />
<s:link source="create:createSessionReturn" sink="finish:sessionID" />
<s:link source="create:createSessionReturn" sink="getresults:sessionID" />
<s:link source="getChildren:getChildrenReturn" sink="add:geneOntologyID" />
```

```
<s:link source="getParents:getParentsReturn" sink="getChildren:geneOntologyID" />
<s:link source="termID" sink="getAncestry:geneOntologyID" />
<s:link source="create:createSessionReturn" sink="markAncestors:sessionID" />
<s:link source="getAncestry:getAncestorsReturn" sink="markAncestors:geneOntologyID" />
<s:link source="termID" sink="getImmediateChildren:geneOntologyID" />
<s:link source="childColour:value" sink="colourChildren:colour" />
<s:link source="create:createSessionReturn" sink="addImmediateChildren:sessionID" />
<s:link source="create:createSessionReturn" sink="colourChildren:sessionID" />
<s:link source="getImmediateChildren:getChildrenReturn" sink="addImmediateChildren:geneOntologyID" />
<s:link source="getImmediateChildren:getChildrenReturn" sink="colourChildren:geneOntologyID" />
<s:link source="inputTermColour:value" sink="colourInputTerm:colour" />
<s:link source="termID" sink="colourInputTerm:geneOntologyID" />
<s:link source="create:createSessionReturn" sink="colourInputTerm:sessionID" />
<s:link source="getresults:getDotReturn" sink="graphical" />
<s:source name="termID">
  <s:metadata>
    <s:description>The term for which to display context within the Gene Ontology.
    An example is the term for vision, GO:0007601</s:description>
  </s:metadata>
</s:source>
<s:sink name="graphical">
  <s:metadata>
    <s:mimeTypes>
      <s:mimeType>text/x-graphviz</s:mimeType>
    </s:mimeTypes>
    <s:description>A text file containing the dot coded representation of the GO subgraph generated
    by this workflow. To see the graph you'll have to run it through the 'dot' tool, part of AT&amp;amp;T's
    graphviz package.</s:description>
  </s:metadata>
</s:sink>
<s:coordination name="finish_BLOCKON_getresults">
  <s:condition>
    <s:state>Completed</s:state>
    <s:target>getresults</s:target>
  </s:condition>
  <s:action>
    <s:target>finish</s:target>
    <s:statechange>
      <s:from>Scheduled</s:from>
      <s:to>Running</s:to>
    </s:statechange>
  </s:action>
</s:coordination>
<s:coordination name="markAncestors_BLOCKON_add">
  <s:condition>
    <s:state>Completed</s:state>
    <s:target>add</s:target>
  </s:condition>
  <s:action>
    <s:target>markAncestors</s:target>
    <s:statechange>
      <s:from>Scheduled</s:from>
      <s:to>Running</s:to>
    </s:statechange>
  </s:action>
</s:coordination>
<s:coordination name="colourChildren_BLOCKON_addImmediateChildren">
  <s:condition>
    <s:state>Completed</s:state>
    <s:target>addImmediateChildren</s:target>
  </s:condition>
  <s:action>
```

```
      <s:target>colourChildren</s:target>
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
  <s:coordination name="getresults_BLOCKON_colourChildren">
    <s:condition>
      <s:state>Completed</s:state>
      <s:target>colourChildren</s:target>
    </s:condition>
    <s:action>
      <s:target>getresults</s:target>
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
  <s:coordination name="colourInputTerm_BLOCKON_markAncestors">
    <s:condition>
      <s:state>Completed</s:state>
      <s:target>markAncestors</s:target>
    </s:condition>
    <s:action>
      <s:target>colourInputTerm</s:target>
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
  <s:coordination name="getresults_BLOCKON_colourInputTerm">
    <s:condition>
      <s:state>Completed</s:state>
      <s:target>colourInputTerm</s:target>
    </s:condition>
    <s:action>
      <s:target>getresults</s:target>
      <s:statechange>
        <s:from>Scheduled</s:from>
        <s:to>Running</s:to>
      </s:statechange>
    </s:action>
  </s:coordination>
</s:scufl>
```

LCC interaction models for multiple agents as generated from translation algorithm for the “Show Gene Ontology Context” workflow are given below:

```
a(ancestorColour, ID) ::
((doneTransfer(ancestorColour:value) => a(markAncestors, ID))) <- perform([], [ancestorColour:value])


a(getChildren, ID) ::
((transfer(getParents:getParentsReturn, getChildren:geneOntologyID)) <-
   (doneTransfer(getParents:getParentsReturn) <= a(getParents, ID)))
then
((doneTransfer(getChildren:getChildrenReturn) => a(add, ID))) <-
   perform([getChildren:geneOntologyID], [getChildren:getChildrenReturn])
```

```
a(getAncestry, ID) ::
((transfer(system:termID, getAncestry:geneOntologyID)) <-
   (doneTransfer(system:termID) <= a(system, ID)))
then
((doneTransfer(getAncestry:getAncestorsReturn) => a(markAncestors, ID))) <-
   perform([getAncestry:geneOntologyID], [getAncestry:getAncestorsReturn])


a(getParents, ID) ::
((transfer(system:termID, getParents:geneOntologyID)) <-
   (doneTransfer(system:termID) <= a(system, ID)))
then
((doneTransfer(getParents:getParentsReturn) => a(getChildren, ID))) <-
   perform([getParents:geneOntologyID], [getParents:getParentsReturn])


a(inputTermColour, ID) ::
((doneTransfer(inputTermColour:value) => a(colourInputTerm, ID))) <-
   perform([], [inputTermColour:value])


a(colourInputTerm, ID) ::
(go() <= a(markAncestors, ID))
then
((transfer(create:createSessionReturn, colourInputTerm:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((transfer(system:termID, colourInputTerm:geneOntologyID)) <-
   (doneTransfer(system:termID) <= a(system, ID)))
then
((transfer(inputTermColour:value, colourInputTerm:colour)) <-
   (doneTransfer(inputTermColour:value) <= a(inputTermColour, ID)))
then
((go() => a(getresults, ID))) <-
   perform([colourInputTerm:geneOntologyID, colourInputTerm:sessionID,
               colourInputTerm:colour], [])


a(getImmediateChildren, ID) ::
((transfer(system:termID, getImmediateChildren:geneOntologyID)) <-
   (doneTransfer(system:termID) <= a(system, ID)))
then
((doneTransfer(getImmediateChildren:getChildrenReturn) => a(addImmediateChildren, ID))
then
(doneTransfer(getImmediateChildren:getChildrenReturn) => a(colourChildren, ID))) <-
   perform([getImmediateChildren:geneOntologyID], [getImmediateChildren:getChildrenReturn])


a(markAncestors, ID) ::
(go() <= a(add, ID))
then
((transfer(getAncestry:getAncestorsReturn, markAncestors:geneOntologyID)) <-
   (doneTransfer(getAncestry:getAncestorsReturn) <= a(getAncestry, ID)))
then
((transfer(create:createSessionReturn, markAncestors:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((transfer(ancestorColour:value, markAncestors:colour)) <-
   (doneTransfer(ancestorColour:value) <= a(ancestorColour, ID)))
then
((go() => a(colourInputTerm, ID))) <-
```

```
   perform([markAncestors:geneOntologyID, markAncestors:colour, markAncestors:sessionID], [])


a(getresults, ID) ::
(go() <= a(colourChildren, ID))
then
(go() <= a(colourInputTerm, ID))
then
((transfer(create:createSessionReturn, getresults:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((doneTransfer(getresults:getDotReturn) => a(system, ID))
then
(go() => a(finish, ID))) <-
   perform([getresults:sessionID], [getresults:getDotReturn])


a(finish, ID) ::
(go() <= a(getresults, ID))
then
((transfer(create:createSessionReturn, finish:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
null <- perform([finish:sessionID], [])


a(add, ID) ::
((transfer(getChildren:getChildrenReturn, add:geneOntologyID)) <-
   (doneTransfer(getChildren:getChildrenReturn) <= a(getChildren, ID)))
then
((transfer(create:createSessionReturn, add:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((go() => a(markAncestors, ID))) <- perform([add:sessionID, add:geneOntologyID], [])


a(addImmediateChildren, ID) ::
((transfer(getImmediateChildren:getChildrenReturn, addImmediateChildren:geneOntologyID)) <-
   (doneTransfer(getImmediateChildren:getChildrenReturn) <= a(getImmediateChildren, ID)))
then
((transfer(create:createSessionReturn, addImmediateChildren:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((go() => a(colourChildren, ID))) <-
   perform([addImmediateChildren:geneOntologyID, addImmediateChildren:sessionID], [])


a(childColour, ID) ::
((doneTransfer(childColour:value) => a(colourChildren, ID))) <- perform([], [childColour:value])


a(colourChildren, ID) ::
(go() <= a(addImmediateChildren, ID))
then
((transfer(getImmediateChildren:getChildrenReturn, colourChildren:geneOntologyID)) <-
   (doneTransfer(getImmediateChildren:getChildrenReturn) <= a(getImmediateChildren, ID)))
then
((transfer(create:createSessionReturn, colourChildren:sessionID)) <-
   (doneTransfer(create:createSessionReturn) <= a(create, ID)))
then
((transfer(childColour:value, colourChildren:colour)) <-
   (doneTransfer(childColour:value) <= a(childColour, ID)))
```

```
then
((go() => a(getresults, ID))) <-
   perform([colourChildren:geneOntologyID, colourChildren:sessionID, colourChildren:colour], [])


a(create, ID) ::
((doneTransfer(create:createSessionReturn) => a(add, ID))
then
(doneTransfer(create:createSessionReturn) => a(finish, ID))
then
(doneTransfer(create:createSessionReturn) => a(getresults, ID))
then
(doneTransfer(create:createSessionReturn) => a(markAncestors, ID))
then
(doneTransfer(create:createSessionReturn) => a(addImmediateChildren, ID))
then
(doneTransfer(create:createSessionReturn) => a(colourChildren, ID))
then
(doneTransfer(create:createSessionReturn) => a(colourInputTerm, ID))) <-
   perform([], [create:createSessionReturn])


a(system, ID) ::
(doneTransfer(system:termID) => a(getParents, ID))
then
(doneTransfer(system:termID) => a(getAncestry, ID))
then
(doneTransfer(system:termID) => a(getImmediateChildren, ID))
then
(doneTransfer(system:termID) => a(colourInputTerm, ID))
then
((transfer(getresults:getDotReturn, system:graphical)) <-
   (doneTransfer(getresults:getDotReturn) <= a(getresults, ID)))
```

A.3.3 BioMart and EMBOSS Analysis Workflow Example:

This Taverna workflow example uses BioMart and EMBOSS services for analysis. Figure A.3 shows workflow graph as rendered by the Taverna.

The SCUFL script for "BioMart And EMBOSS Analysis" workflow is listed below:

```
<?xml version="1.0" encoding="UTF-8"?>
<s:scufl xmlns:s="http://org.embl.ebi.escience/xscufl/0.1alpha" version="0.2" log="0">
  <s:workflowdescription lsid="urn:lsid:www.mygrid.org.uk:operation:E9TFHN0YVY0" author="" title="BiomartAndEMBOSSAnaly
  <s:processor name="FlattenImageList">
    <s:local>org.embl.ebi.escience.scuflworkers.java.FlattenList</s:local>
  </s:processor>
  <s:processor name="CreateFasta">
    <s:beanshell>
      <s:scriptvalue>fasta = "&gt;Human\n"+hsSeq+"\n&gt;Mouse\n"+mmSeq+"\n&gt;Rat\n"+rnSeq;</s:scriptvalue>
      <s:beanshellinputlist>
        <s:beanshellinput s:syntactictype="'text/plain'">hsSeq</s:beanshellinput>
        <s:beanshellinput s:syntactictype="'text/plain'">mmSeq</s:beanshellinput>
        <s:beanshellinput s:syntactictype="'text/plain'">rnSeq</s:beanshellinput>
      </s:beanshellinputlist>
      <s:beanshelloutputlist>
        <s:beanshelloutput s:syntactictype="'text/plain'">fasta</s:beanshelloutput>
      </s:beanshelloutputlist>
    </s:beanshell>
```

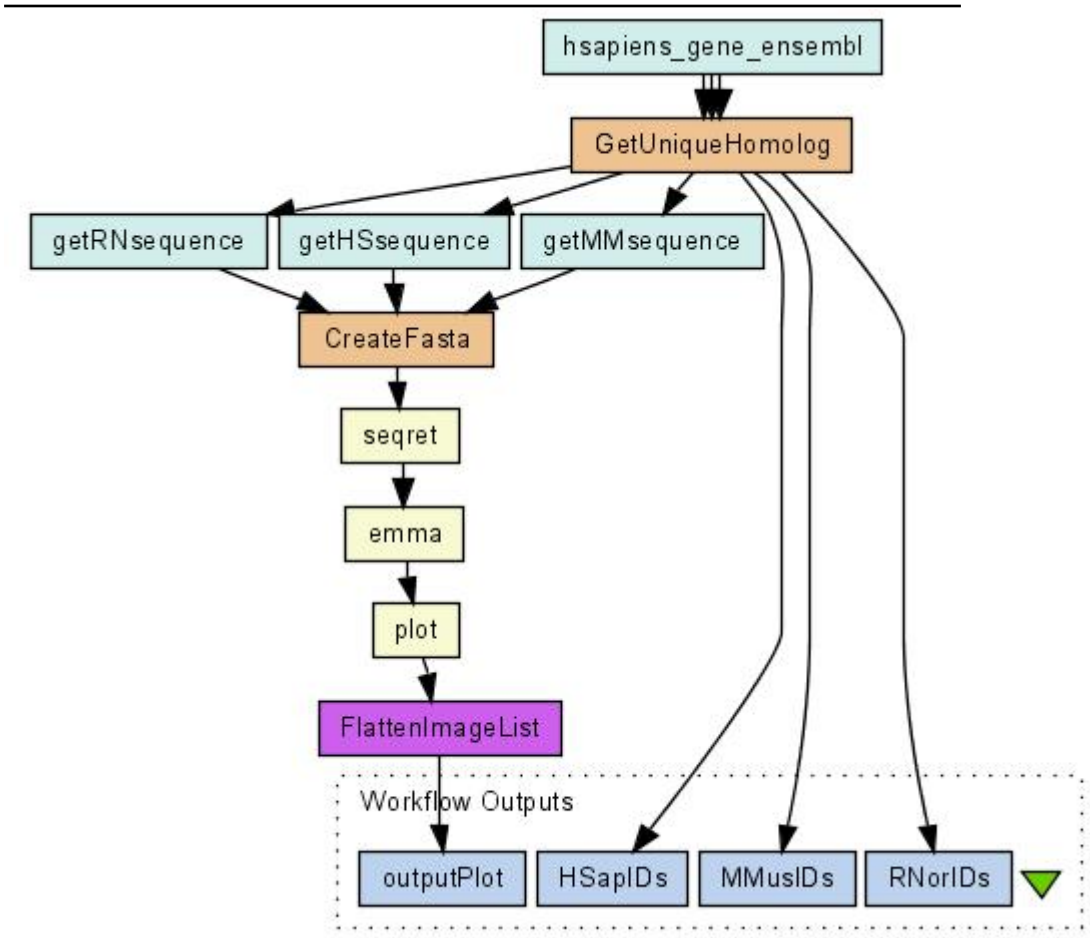

Figure A.3: Taverna Workflow to show Gene Ontology Context for supplied terms

```
    <s:iterationstrategy>
      <i:dot xmlns:i="http://org.embl.ebi.escience/xscufliteration/0.1beta10">
        <i:iterator name="rnSeq" />
        <i:iterator name="mmSeq" />
        <i:iterator name="hsSeq" />
      </i:dot>
    </s:iterationstrategy>
  </s:processor>
  <s:processor name="GetUniqueHomolog">
    <s:beanshell>
      <s:scriptvalue>
List HSOut = new ArrayList();
List RatOut = new ArrayList();
List MouseOut = new ArrayList();

Map hsToMouse = new HashMap();
Iterator j = MouseGeneIDs.iterator();
for (Iterator i = HSGeneIDs.iterator(); i.hasNext();) {
  String id = (String)i.next();
  hsToMouse.put(id, j.next());
}
Map hsToRat = new HashMap();
j = RatGeneIDs.iterator();
for (Iterator i = HSGeneIDs.iterator(); i.hasNext();) {
  String id = (String)i.next();
  hsToRat.put(id, j.next());
}

// Build the unique outputs
for (Iterator i = hsToRat.keySet().iterator(); i.hasNext();) {
  String hsID = (String)i.next();
  String ratID = (String)hsToRat.get(hsID);
  // Remove version number
  // ratID = (ratID.split("."))[0];
  String mouseID = (String)hsToMouse.get(hsID);
  // Remove version number
  //mouseId = (mouseID.split("."))[0];
  if (ratID != null &amp;amp;&amp;amp; mouseID != null &amp;amp;&amp;amp;
    ratID.equals("")==false &amp;amp;&amp;amp; mouseID.equals("")==false) {
    HSOut.add(hsID);
    RatOut.add(ratID.split("\\.")[0]);
    MouseOut.add(mouseID.split("\\.")[0]);
  }
}</s:scriptvalue>
      <s:beanshellinputlist>
        <s:beanshellinput s:syntactictype="l('text/plain')">HSGeneIDs</s:beanshellinput>
        <s:beanshellinput s:syntactictype="l('text/plain')">MouseGeneIDs</s:beanshellinput>
        <s:beanshellinput s:syntactictype="l('text/plain')">RatGeneIDs</s:beanshellinput>
      </s:beanshellinputlist>
      <s:beanshelloutputlist>
        <s:beanshelloutput s:syntactictype="l('text/plain')">HSOut</s:beanshelloutput>
        <s:beanshelloutput s:syntactictype="l('text/plain')">RatOut</s:beanshelloutput>
        <s:beanshelloutput s:syntactictype="l('text/plain')">MouseOut</s:beanshelloutput>
      </s:beanshelloutputlist>
    </s:beanshell>
  </s:processor>
  <s:processor name="seqret" workers="5">
    <s:description>Reads and writes (returns) sequences</s:description>
    <s:soaplabwsdl>http://www.ebi.ac.uk/soaplab/services/edit.seqret</s:soaplabwsdl>
  </s:processor>
  <s:processor name="plot" workers="5">
    <s:description>Displays aligned sequences, with colouring and boxing</s:description>
```

```
  <s:soaplabwsdl>http://www.ebi.ac.uk/soaplab/services/alignment_multiple.prettyplot</s:soaplabwsdl>
</s:processor>
<s:processor name="emma" workers="5">
  <s:description>Multiple alignment program - interface to ClustalW program</s:description>
  <s:soaplabwsdl>http://www.ebi.ac.uk/soaplab/services/alignment_multiple.emma</s:soaplabwsdl>
</s:processor>
<s:processor name="getRNsequence">
  <s:description>Rattus norvegicus genes (RGSC3.4)</s:description>
  <s:biomart>
    <biomart:MartQuery xmlns:biomart="http://org.embl.ebi.escience/xscufl-biomart/0.1alpha">
      <biomart:MartService location="http://www.biomart.org/biomart/martservice" />
      <biomart:MartDataset displayName="Rattus norvegicus genes (RGSC3.4)"
      name="rnorvegicus_gene_ensembl" type="TableSet" initialBatchSize="10"
      maximumBatchSize="50000" visible="false" interface="default" modified="2006-11-27 12:39:16">
      <biomart:MartURLLocation database="ensembl_mart_41" default="1" displayName="ENSEMBL 41  (SANGER)"
       host="www.biomart.org" includeDatasets="" martUser="" name="ensembl"
       path="/biomart/martservice" port="80" serverVirtualSchema="default"
       virtualSchema="default" visible="1" />
      </biomart:MartDataset>
      <biomart:Query virtualSchemaName="default" count="0" softwareVersion="0.5" requestId="taverna">
        <biomart:Dataset name="rnorvegicus_gene_ensembl">
          <biomart:Attribute name="gene_stable_id" />
          <biomart:Attribute name="coding_gene_flank" />
          <biomart:Filter name="ensembl_gene_id" value="" list="true" />
          <biomart:Filter name="upstream_flank" value="100" />
        </biomart:Dataset>
      </biomart:Query>
    </biomart:MartQuery>
  </s:biomart>
</s:processor>
<s:processor name="getHSsequence">
  <s:description>Homo sapiens genes (NCBI36)</s:description>
  <s:biomart>
    <biomart:MartQuery xmlns:biomart="http://org.embl.ebi.escience/xscufl-biomart/0.1alpha">
      <biomart:MartService location="http://www.biomart.org/biomart/martservice" />
      <biomart:MartDataset displayName="Homo sapiens genes (NCBI36)" name="hsapiens_gene_ensembl"
      type="TableSet" initialBatchSize="10" maximumBatchSize="50000" visible="false"
      interface="default" modified="2006-11-27 12:49:27">
      <biomart:MartURLLocation database="ensembl_mart_41" default="1" displayName="ENSEMBL 41  (SANGER)"
       host="www.biomart.org" includeDatasets="" martUser="" name="ensembl"
       path="/biomart/martservice" port="80" serverVirtualSchema="default"
       virtualSchema="default" visible="1" />
      </biomart:MartDataset>
      <biomart:Query virtualSchemaName="default" count="0" softwareVersion="0.5" requestId="taverna">
        <biomart:Dataset name="hsapiens_gene_ensembl">
          <biomart:Attribute name="coding_gene_flank" />
          <biomart:Attribute name="gene_stable_id" />
          <biomart:Filter name="ensembl_gene_id" value="" list="true" />
          <biomart:Filter name="upstream_flank" value="100" />
        </biomart:Dataset>
      </biomart:Query>
    </biomart:MartQuery>
  </s:biomart>
</s:processor>
<s:processor name="getMMsequence">
  <s:description>Mus musculus genes (NCBIM36)</s:description>
  <s:biomart>
    <biomart:MartQuery xmlns:biomart="http://org.embl.ebi.escience/xscufl-biomart/0.1alpha">
      <biomart:MartService location="http://www.biomart.org/biomart/martservice" />
      <biomart:MartDataset displayName="Mus musculus genes (NCBIM36)" name="mmusculus_gene_ensembl"
       type="TableSet" initialBatchSize="10" maximumBatchSize="50000" visible="false"
       interface="default" modified="2006-11-27 12:45:14">
```

```
      <biomart:MartURLLocation database="ensembl_mart_41" default="1" displayName="ENSEMBL 41  (SANGER)"
       host="www.biomart.org" includeDatasets="" martUser="" name="ensembl"
       path="/biomart/martservice" port="80" serverVirtualSchema="default"
       virtualSchema="default" visible="1" />
      </biomart:MartDataset>
      <biomart:Query virtualSchemaName="default" count="0" softwareVersion="0.5" requestId="taverna">
        <biomart:Dataset name="mmusculus_gene_ensembl">
          <biomart:Attribute name="coding_gene_flank" />
          <biomart:Attribute name="gene_stable_id" />
          <biomart:Filter name="ensembl_gene_id" value="" list="true" />
          <biomart:Filter name="upstream_flank" value="100" />
        </biomart:Dataset>
      </biomart:Query>
    </biomart:MartQuery>
  </s:biomart>
</s:processor>
<s:processor name="hsapiens_gene_ensembl">
  <s:description>Homo sapiens genes (NCBI36)</s:description>
  <s:biomart>
    <biomart:MartQuery xmlns:biomart="http://org.embl.ebi.escience/xscufl-biomart/0.1alpha">
    <biomart:MartService location="http://www.biomart.org/biomart/martservice" />
    <biomart:MartDataset displayName="Homo sapiens genes (NCBI36)" name="hsapiens_gene_ensembl"
    type="TableSet" initialBatchSize="10" maximumBatchSize="50000" visible="false"
    interface="default" modified="2006-11-27 12:49:27">
    <biomart:MartURLLocation database="ensembl_mart_41" default="1" displayName="ENSEMBL 41  (SANGER)"
     host="www.biomart.org" includeDatasets="" martUser="" name="ensembl"
     path="/biomart/martservice" port="80" serverVirtualSchema="default"
     virtualSchema="default" visible="1" />
    </biomart:MartDataset>
    <biomart:Query virtualSchemaName="default" count="0" softwareVersion="0.5" requestId="taverna">
    <biomart:Dataset name="hsapiens_gene_ensembl">
        <biomart:Attribute name="ensembl_gene_id" />
        <biomart:Attribute name="mouse_ensembl_gene" />
        <biomart:Attribute name="rat_ensembl_gene" />
        <biomart:Filter name="chromosome_name" value="22" />
          <biomart:Filter name="with_mmusculus_homolog" excluded="0" />
          <biomart:Filter name="with_mim_morbid" excluded="0" />
        </biomart:Dataset>
      </biomart:Query>
    </biomart:MartQuery>
  </s:biomart>
</s:processor>
<s:link source="CreateFasta:fasta" sink="seqret:sequence_direct_data" />
<s:link source="GetUniqueHomolog:HSOut"
 sink="getHSsequence:hsapiens_gene_ensembl.ensembl_gene_id_filter" />
<s:link source="GetUniqueHomolog:MouseOut"
 sink="getMMsequence:mmusculus_gene_ensembl.ensembl_gene_id_filter" />
<s:link source="GetUniqueHomolog:RatOut"
 sink="getRNsequence:rnorvegicus_gene_ensembl.ensembl_gene_id_filter" />
<s:link source="emma:outseq" sink="plot:sequences_direct_data" />
<s:link source="getHSsequence:hsapiens_gene_ensembl.coding_gene_flank"
 sink="CreateFasta:hsSeq" />
<s:link source="getMMsequence:mmusculus_gene_ensembl.coding_gene_flank"
 sink="CreateFasta:mmSeq" />
<s:link source="getRNsequence:rnorvegicus_gene_ensembl.coding_gene_flank"
 sink="CreateFasta:rnSeq" />
<s:link source="hsapiens_gene_ensembl:hsapiens_gene_ensembl.ensembl_gene_id"
 sink="GetUniqueHomolog:HSGeneIDs" />
<s:link source="hsapiens_gene_ensembl:hsapiens_gene_ensembl.mouse_ensembl_gene"
 sink="GetUniqueHomolog:MouseGeneIDs" />
<s:link source="hsapiens_gene_ensembl:hsapiens_gene_ensembl.rat_ensembl_gene"
 sink="GetUniqueHomolog:RatGeneIDs" />
```

```
  <s:link source="plot:Graphics_in_PNG" sink="FlattenImageList:inputlist" />
  <s:link source="FlattenImageList:outputlist" sink="outputPlot" />
  <s:link source="GetUniqueHomolog:HSOut" sink="HSapIDs" />
  <s:link source="GetUniqueHomolog:MouseOut" sink="MMusIDs" />
  <s:link source="GetUniqueHomolog:RatOut" sink="RNorIDs" />
  <s:link source="seqret:outseq" sink="emma:sequence_direct_data" />
  <s:sink name="outputPlot">
    <s:metadata>
      <s:mimeTypes>
        <s:mimeType>image/png</s:mimeType>
        <s:mimeType>application/octet-stream</s:mimeType>
      </s:mimeTypes>
      <s:description>The array of png images returned from the plot processor</s:description>
      <s:semanticType>http://www.mygrid.org.uk/ontology#domain_concept</s:semanticType>
    </s:metadata>
  </s:sink>
  <s:sink name="HSapIDs" />
  <s:sink name="MMusIDs" />
  <s:sink name="RNorIDs" />
</s:scufl>
```

LCC interaction models for multiple agents as generated from translation algorithm for the "BioMart And EMBOSS Analysis" workflow are given below:

```
a(FlattenImageList, ID) ::
((transfer(plot:Graphics_in_PNG, FlattenImageList:inputlist)) <-
   (doneTransfer(plot:Graphics_in_PNG) <= a(plot, ID)))
then
((doneTransfer(FlattenImageList:outputlist) => a(system, ID))) <-

   perform([FlattenImageList:inputlist], [FlattenImageList:outputlist])


a(CreateFasta, ID) ::
((transfer(getRNsequence:rnorvegicus_gene_ensembl.coding_gene_flank, CreateFasta:rnSeq)) <-
  (doneTransfer(getRNsequence:rnorvegicus_gene_ensembl.coding_gene_flank) <= a(getRNsequence, ID)))
then
((transfer(getMMsequence:mmusculus_gene_ensembl.coding_gene_flank, CreateFasta:mmSeq)) <-
  (doneTransfer(getMMsequence:mmusculus_gene_ensembl.coding_gene_flank) <= a(getMMsequence, ID)))
then
((transfer(getHSsequence:hsapiens_gene_ensembl.coding_gene_flank, CreateFasta:hsSeq)) <-
  (doneTransfer(getHSsequence:hsapiens_gene_ensembl.coding_gene_flank) <= a(getHSsequence, ID)))
then
((doneTransfer(CreateFasta:fasta) => a(seqret, ID))) <-
  perform([CreateFasta:hsSeq, CreateFasta:mmSeq, CreateFasta:rnSeq], [CreateFasta:fasta])


a(GetUniqueHomolog, ID) ::
((transfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.rat_ensembl_gene, GetUniqueHomolog:RatGeneIDs)) <-
  (doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.rat_ensembl_gene) <= a(hsapiens_gene_ensembl, I
then
((transfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.mouse_ensembl_gene, GetUniqueHomolog:MouseGeneIDs))
  (doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.mouse_ensembl_gene)
  <= a(hsapiens_gene_ensembl, ID)))
then
((transfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.ensembl_gene_id, GetUniqueHomolog:HSGeneIDs)) <-
  (doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.ensembl_gene_id) <= a(hsapiens_gene_ensembl, ID
then
((doneTransfer(GetUniqueHomolog:HSOut) => a(getHSsequence, ID))
then
(doneTransfer(GetUniqueHomolog:MouseOut) => a(getMMsequence, ID))
```

```
then
(doneTransfer(GetUniqueHomolog:RatOut) => a(getRNsequence, ID))
then
(doneTransfer(GetUniqueHomolog:HSOut) => a(system, ID))
then
(doneTransfer(GetUniqueHomolog:MouseOut) => a(system, ID))
then
(doneTransfer(GetUniqueHomolog:RatOut) => a(system, ID))) <-
   perform([GetUniqueHomolog:HSGeneIDs, GetUniqueHomolog:RatGeneIDs, GetUniqueHomolog:MouseGeneIDs],
          [GetUniqueHomolog:MouseOut, GetUniqueHomolog:HSOut, GetUniqueHomolog:RatOut])


a(seqret, ID) ::
((transfer(CreateFasta:fasta, seqret:sequence_direct_data)) <- (doneTransfer(CreateFasta:fasta)
  <= a(CreateFasta, ID)))
then
((doneTransfer(seqret:outseq) => a(emma, ID))) <- perform([seqret:sequence_direct_data], [seqret:outseq])


a(plot, ID) ::
((transfer(emma:outseq, plot:sequences_direct_data)) <- (doneTransfer(emma:outseq) <= a(emma, ID)))
then
((doneTransfer(plot:Graphics_in_PNG) => a(FlattenImageList, ID))) <-
  perform([plot:sequences_direct_data], [plot:Graphics_in_PNG])


a(emma, ID) ::
((transfer(seqret:outseq, emma:sequence_direct_data)) <- (doneTransfer(seqret:outseq) <= a(seqret, ID)))
then
((doneTransfer(emma:outseq) => a(plot, ID))) <-
  perform([emma:sequence_direct_data], [emma:outseq])


a(getRNsequence, ID) ::
((transfer(GetUniqueHomolog:RatOut, getRNsequence:rnorvegicus_gene_ensembl.ensembl_gene_id_filter)) <-
  (doneTransfer(GetUniqueHomolog:RatOut) <= a(GetUniqueHomolog, ID)))
then
((doneTransfer(getRNsequence:rnorvegicus_gene_ensembl.coding_gene_flank) => a(CreateFasta, ID))) <-
  perform([getRNsequence:rnorvegicus_gene_ensembl.ensembl_gene_id_filter],
          [getRNsequence:rnorvegicus_gene_ensembl.coding_gene_flank])


a(getHSsequence, ID) ::
((transfer(GetUniqueHomolog:HSOut, getHSsequence:hsapiens_gene_ensembl.ensembl_gene_id_filter)) <-
  (doneTransfer(GetUniqueHomolog:HSOut) <= a(GetUniqueHomolog, ID)))
then
((doneTransfer(getHSsequence:hsapiens_gene_ensembl.coding_gene_flank) => a(CreateFasta, ID))) <-
    perform([getHSsequence:hsapiens_gene_ensembl.ensembl_gene_id_filter],
            [getHSsequence:hsapiens_gene_ensembl.coding_gene_flank])


a(getMMsequence, ID) ::
((transfer(GetUniqueHomolog:MouseOut, getMMsequence:mmusculus_gene_ensembl.ensembl_gene_id_filter)) <-
  (doneTransfer(GetUniqueHomolog:MouseOut) <= a(GetUniqueHomolog, ID)))
then
((doneTransfer(getMMsequence:mmusculus_gene_ensembl.coding_gene_flank) => a(CreateFasta, ID))) <-
   perform([getMMsequence:mmusculus_gene_ensembl.ensembl_gene_id_filter],
           [getMMsequence:mmusculus_gene_ensembl.coding_gene_flank])


a(hsapiens_gene_ensembl, ID) ::
((doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.ensembl_gene_id) => a(GetUniqueHomolog, ID))
```

```
then
(doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.mouse_ensembl_gene) => a(GetUniqueHomolog, ID))
then
(doneTransfer(hsapiens_gene_ensembl:hsapiens_gene_ensembl.rat_ensembl_gene) => a(GetUniqueHomolog, ID))) <-
   perform([], [hsapiens_gene_ensembl:hsapiens_gene_ensembl.mouse_ensembl_gene,
                hsapiens_gene_ensembl:hsapiens_gene_ensembl.ensembl_gene_id,
                hsapiens_gene_ensembl:hsapiens_gene_ensembl.rat_ensembl_gene])


a(system, ID) ::
((transfer(GetUniqueHomolog:RatOut, system:RNorIDs)) <-
   (doneTransfer(GetUniqueHomolog:RatOut) <= a(GetUniqueHomolog, ID)))
then
((transfer(GetUniqueHomolog:MouseOut, system:MMusIDs)) <-
   (doneTransfer(GetUniqueHomolog:MouseOut) <= a(GetUniqueHomolog, ID)))
then
((transfer(GetUniqueHomolog:HSOut, system:HSapIDs)) <-
   (doneTransfer(GetUniqueHomolog:HSOut) <= a(GetUniqueHomolog, ID)))
then
((transfer(FlattenImageList:outputlist, system:outputPlot)) <-
   (doneTransfer(FlattenImageList:outputlist) <= a(FlattenImageList, ID)))
```

A.4 Explicit Iteration Strategy Prolog Script

The following Prolog script can parse the explicit iteration strategy statement and then generate respective service calls using input parameter items. The results for the generated service calls are then returned to the script component caller. *"iterate_main"* is the main rule for this Prolog script.

```
%Main rule to handle the Specific Iteration call
iterate_main(Inputs, Service_Name, Results):-
   iterate(Inputs, Outputs),
   flatten_output(Outputs, Parameters),
   convert_to_calls(Parameters, Service_Name, Calls),
   make_calls(Calls, Results).

iterate(Inputs, Output):-
   Inputs =.. [Operation, List],
   iterate(List, Output, Operation).

is_flat([]).
is_flat([H | T]):-
   is_list(H),
   is_flat(T).

iterate(Inputs, Output, Operation):-
   is_flat(Inputs),
   Operation \= delay,
   generate(Inputs, Output, Operation).
```

```
iterate(Inputs, Inputs, delay):-
   is_flat(Inputs).

iterate([HI | TI], Output, Operation):-
   \+ is_list(HI),
   HI =.. [HOperation, HList],
   iterate(HList, HOutput, HOperation),
   iterate(TI, TOutput, delay),
   append(HOutput, TOutput, List),
   iterate(List, Output, Operation).

iterate([HI | TI], Output, Operation):-
   is_list(HI),
   iterate(TI, TOutput, delay),
   iterate([HI | TOutput], Output, Operation).

generate(Inputs, [Output], cross):-
   setof(O, select_inputs(Inputs, O), Output).

generate(Inputs, [Output], dot):-
   setof(O, select_inputs_dot(Inputs, O), Output).

select_inputs([L | T], [X | R]):-
   member(X, L),
   select_inputs(T, R).

select_inputs([], []).

member_n(0, H, [H | _]).

member_n(N, H, [_ | T]):-
   member_n(M, H, T),
   N is M + 1.

length([], 0).
length([_ | T], N):-
   length(T, M),
   N is M + 1.

get_I(L, N):-
   get_I(L, 0, N).

get_I(L, N, N):-
   N < L.
```

```
get_I(L, M, N):-
   G is M + 1,
   G < L,
   get_I(L, G, N).

select_inputs_dot(Input, Output):-
   Input = [H | _],
   length(H, L),
   get_I(L, I),
   select_inputs_d(Input, Output, I).

select_inputs_d([H | T], [X | R], I):-
   member_n(I, X, H),
   select_inputs_d(T, R, I).

select_inputs_d([], [], _).

flatten_list(List, Flat):-
   flatten_list(List, Flat, []).

flatten_list([H|T]) --> flatten_list(H), flatten_list(T).
flatten_list([]) --> !, [].
flatten_list(H) --> {atomic(H)}, !, [H].

flatten([], []).
flatten([H | T], [FH | FT]):-
   flatten_list(H, FH),
   flatten(T, FT).

%rule to flatten the parameters combinations generated.
flatten_output([ L ], FL):-
   flatten(L, FL).

%rule to generate the calls from flattened parameters and service name.
convert_to_calls([], _, []).
convert_to_calls([H | T], Service_Name, [c(CH, O) | CT]):-
   append(H, [O], Par),
   CH =.. [Service_Name | Par],
   convert_to_calls(T, Service_Name, CT).

%rules to execute the calls and generate the results to be returned to the call
make_calls([], []).
make_calls([c(H, HO) | T], [HO | TO]):-
```

```
  H,
  make_calls(T, TO).
```

A.5 Basic Translation Algorithm Implementation Source Code

Following is the source code for the java implementation of the basic translation algorithm:

A.5.1 Translator.java

```
/**
 *  Class: Translator
  * Description: This class is the main class for the translation algorithm. It
  * takes the file path SCUFL workflow XML file and then generates the LCC
  * interaction model script for the multi agents. This class uses the Reader
  * class to read and extract the required information from the SCUFL workflow
  * file and then applied the translation algorithm to produce an output script.
 *
 */
import java.util.Vector;
import java.util.ListIterator;
import java.util.HashMap;
import org.jdom.Element;
import java.util.Collection;
import java.util.Iterator;

public class Translator {
  public static final String MESS_DONE_TRANSFER = "doneTransfer";
  public static final String MESS_TRANSFER = "transfer";
  public static final String MESS_THEN = "then";
  public static final String MESS_GO = "go";
  public static final String MESS_PERFORM = "perform";
  public Reader objReader;

  /**
   * Constructor
   * @param strDocumentName String = SCUFL workflow file path.
   */
  public Translator(String strDocumentName) {
    objReader = new Reader(strDocumentName);
    //do preprocessing for the system peer.
    objReader.PreProcessingForSystemPeer();
  }

  /**
   * This function would return a vector contain subgroup (vectors)
   * of link statements (elements) with the same sink parts
   * @param list_sink Vector
   * @return Vector
   */
  public Vector groupLinkStatementBasedonSinkValue (Vector list_sink){
    Vector toRet = new Vector();
    if (list_sink.size() > 0)
    {
      while (!list_sink.isEmpty())
      {
```

```
      Vector subgroup = new Vector();
      Element eleLink = (Element)list_sink.remove(list_sink.size()-1);
      subgroup.add (eleLink);
      String tomatch = eleLink.getAttributeValue(Reader.ATT_SINK);
      for (int i = list_sink.size()-1; i >=0; i--)
      {
        Element toTest = (Element)list_sink.elementAt(i);
        String strTestSink = toTest.getAttributeValue(Reader.ATT_SINK);
        if (strTestSink.equals(tomatch))
        {
          //matching sink statement found and thus should be
          //removed from the main group and added to the subgroup
          subgroup.add(toTest);
          list_sink.remove(i);
        }
      }
      toRet.add(subgroup);
    }
    if (toRet.size() > 0)
      return toRet;
    else return null;
  }
  else return null;
}

/**
 * Function to build transfer message from link statement
 * @param linkStatement Element = link element
 * @return String = translated LCC transfer statement
 */
public String buildTransferMessageScriptFromMessage(Element linkStatement){
  String sourceport;     String source;     String sinkport;
  sinkport = linkStatement.getAttributeValue(Reader.ATT_SINK);
  sourceport = linkStatement.getAttributeValue(Reader.ATT_SOURCE);
  String parts[] = sourceport.split(":");
  source = parts[0];

  String toRet = "("+ MESS_TRANSFER.concat( "("+sourceport +", "+ sinkport+")") + ") <- (" +
      MESS_DONE_TRANSFER.concat("("+sourceport+") <= a("+source+", ID))");

  return toRet;
}

/**
 * builds the transfer message script part of the protocol for a processor.
 * @param vecLinkStats Vector
 * @return String
 */
public String getMessageScriptForLinkGroup(Vector vecLinkStats){
  String strMainScript = "";
  for (int i = 0; i < vecLinkStats.size(); i++)
  {
    Element statement = (Element) vecLinkStats.elementAt(i);
    if (i == 0)
      strMainScript = "(" + buildTransferMessageScriptFromMessage(statement) + ")";
    else
      strMainScript = strMainScript + " or (" + buildTransferMessageScriptFromMessage(statement) + ")";
  }
  return strMainScript;
}

/**
```

```
 * Generate incoming message part of the agent protocol script for a processor.
 * @param strProcessorName String = SCUFL Processor name
 * @return String
 */
public String generateIncomingTransferMsgsScript(String strProcessorName){

  //Get link statements with processor as sink
  String strMainScript = "";
  Vector list_sink = objReader.getLinkWithProcessorAsSinkOrSource(strProcessorName, Reader.ATT_SINK);
  if (list_sink != null){
    if (list_sink.size() > 0){
      Vector groups = groupLinkStatementBasedonSinkValue(list_sink);
      if (groups != null)
      {
        for (int i = 0; i < groups.size(); i++){
          Vector subgroup = (Vector)groups.elementAt(i);
          if (i == 0)
            strMainScript = getMessageScriptForLinkGroup(subgroup);
          else
            strMainScript = strMainScript + "\n" + MESS_THEN + "\n" +
                getMessageScriptForLinkGroup(subgroup);
        }
      }
    }
  }
  return strMainScript;
}

/**
 * Builds the outgoing transfer message from the link element
 * @param link Element
 * @return String = translated LCC transfer statement
 */
public String buildOutgoingTransferMessage(Element link){
  String sourceport; String sink;

  sourceport = link.getAttributeValue(Reader.ATT_SOURCE);
  sink = link.getAttributeValue(Reader.ATT_SINK);
  String parts[] = sink.split(":");
  sink = parts[0];

  String strMessage = MESS_DONE_TRANSFER + "("+ sourceport +") => a("+sink+", ID)";
  return strMessage;
}

/**
 * Builds the outgoing transfer message script part for an agent.
 * @param strProcessorName String = SCUFL processor name
 * @return String = translated LCC script for outgoing message
 */
public String generateOutgoingTransferMsgsScript(String strProcessorName){
  String strMainScript = "";
  Vector list_source = objReader.getLinkWithProcessorAsSinkOrSource(strProcessorName, Reader.ATT_SOURCE);
  if (list_source != null) {
    if (list_source.size() > 0){
      for (int i=0; i<list_source.size(); i++){
        Element link = (Element) list_source.elementAt(i);
        if (i == 0)
          strMainScript = "(" + buildOutgoingTransferMessage(link) + ")";
        else
          strMainScript = strMainScript + "\n" + MESS_THEN + "\n" + "(" + buildOutgoingTransferMessage(link) + ")";
      }
```

```
    }
  }
  return strMainScript;
}

public String buildPreconditionMessage(String strProcName)
{
  return MESS_GO + "() <= a(" + strProcName + ", ID)";
}

/**
 * Build precondition part of the script for an agent.
 * @param strProcessorName String
 * @return String
 */
public String generatePreconditionMessageScript(String strProcessorName){
  String strMainScript = "";
  Vector vecProcNames = objReader.getPreconditionProcessorNames(strProcessorName);
  if (vecProcNames != null){
    if (vecProcNames.size() > 0){
      for (int i = 0; i < vecProcNames.size(); i++){
        if (i == 0)
          strMainScript = "(" + buildPreconditionMessage((String)vecProcNames.elementAt(i)) + ")";
        else
          strMainScript = strMainScript + "\n" + MESS_THEN + "\n" +  "(" +
              buildPreconditionMessage((String)vecProcNames.elementAt(i)) + ")";
      }
    }
  }
  return strMainScript;
}

public String buildPostconditionMessage(String strProcName)
{
  return MESS_GO + "() => a(" + strProcName + ", ID)";
}

/**
 * Builds post condition part of the agent protocol script.
 * @param strProcessorName String
 * @return String = translated LCC protocol script
 */
public String generatePostconditionMessageScript(String strProcessorName){
  String strMainScript = "";
  Vector vecProcNames = objReader.getPostConditionProcessorNames(strProcessorName);
  if (vecProcNames != null){
    if (vecProcNames.size() > 0) {
      for (int i = 0; i < vecProcNames.size(); i++){
        if (i == 0)
          strMainScript = "(" + buildPostconditionMessage((String)vecProcNames.elementAt(i)) + ")";
        else
          strMainScript = strMainScript + "\n" + MESS_THEN + "\n" + "(" +
              buildPostconditionMessage((String)vecProcNames.elementAt(i)) + ")";
      }
    }
  }
  return strMainScript;
}

/**
 * Build LCC perform statement parameter list (input/output)
 * @param strProcessorName String = SCUFL processor name
```

```
 * @param strAttType String = (source/sink)
 * @return String = generated LCC perform clause parameter list.
 */
public String getPerformInputOutputPart (String strProcessorName, String strAttType){
  String strInputPart = "";
  HashMap map;
  Vector list_sink = objReader.getLinkWithProcessorAsSinkOrSource(strProcessorName, strAttType);
  if (list_sink != null){
    if (list_sink.size() > 0){
      map = new HashMap();
      for (int i = 0; i < list_sink.size(); i++){
        Element eleLink = (Element) list_sink.get(i);
        String strStrPort = eleLink.getAttributeValue(strAttType);
        map.put(strStrPort, strStrPort);
      }
      if (map.size()>0){
        Collection objCol = map.values();
        Iterator colIter = objCol.iterator();
        int i = 0;
        while (colIter.hasNext()){
          /*String parts[] = ((String)colIter.next()).split(":");*/
          if (i == 0)
            strInputPart = (String) colIter.next();
          else
            strInputPart = strInputPart + ", " + (String) colIter.next();
          i++;
        }
      }
    }
  }
  return strInputPart;
}

/**
 * Function generates the LCC interaction protocol script for a SCUFL processor.
 * @param strProcessorName String = SCUFL Processor name
 * @return String = generated LCC agent protocol defintion for the processor
 */
public String generateProcessorDefinition(String strProcessorName){

  String strAgentDefinition = "";
  if (!strProcessorName.equals(Reader.PROC_SYSTEM)){
    String strPrecondition = generatePreconditionMessageScript(strProcessorName);
    strPrecondition = strPrecondition.trim();
    if (strPrecondition.length() > 0)
      strAgentDefinition = strPrecondition;

    String strIncoming = generateIncomingTransferMsgsScript(strProcessorName);
    strIncoming = strIncoming.trim();
    if (strIncoming.length() > 0) {
      if (strAgentDefinition.trim().length() > 0)
        strAgentDefinition = strAgentDefinition + "\n" + MESS_THEN + "\n" +
            strIncoming;
      else
        strAgentDefinition = strAgentDefinition + strIncoming;
    }

    String stroutgoing = generateOutgoingTransferMsgsScript(strProcessorName);
    stroutgoing = stroutgoing.trim();
    if (stroutgoing.length() > 0) {
      if (strPrecondition.trim().length() > 0 || strIncoming.trim().length() > 0) {
        strAgentDefinition = strAgentDefinition + "\n" + MESS_THEN + "\n";
```

```
      }
      strAgentDefinition = strAgentDefinition + "(" + stroutgoing;
    }
    //get input and output arguments for the perform function
    String inputArgs = getPerformInputOutputPart(strProcessorName, Reader.ATT_SINK);
    String outputArgs = getPerformInputOutputPart(strProcessorName, Reader.ATT_SOURCE);

    String strpostcondition = generatePostconditionMessageScript(strProcessorName);
    strpostcondition = strpostcondition.trim();
    if (strpostcondition.length() > 0) {
      if (strPrecondition.trim().length() > 0 || strIncoming.trim().length() > 0 ||
          stroutgoing.trim().length() > 0) {
        strAgentDefinition = strAgentDefinition + "\n" + MESS_THEN + "\n";
      }
      if (stroutgoing.trim().length() > 0)
        strAgentDefinition = strAgentDefinition + strpostcondition + ") <- " + MESS_PERFORM + "(["+ input
      else
        strAgentDefinition = strAgentDefinition + "(" + strpostcondition + ") <- " + MESS_PERFORM + "(["+
    }
    else if (stroutgoing.trim().length() > 0)
        strAgentDefinition = strAgentDefinition + ") <- " + MESS_PERFORM + "(["+ inputArgs +"], ["+output

    //if there is no outgoing messages and post condition then
    if (stroutgoing.trim().length() == 0 && strpostcondition.trim().length() == 0)
    {
      if (strPrecondition.trim().length() > 0 || strIncoming.trim().length() > 0)
        strAgentDefinition = strAgentDefinition + "\n" + MESS_THEN + "\n";
      strAgentDefinition = strAgentDefinition  + "null <- " + MESS_PERFORM + "(["+ inputArgs +"], ["+outp
    }
  }
  else
  {
    //build definition for the system processor.
    String stroutgoing = generateOutgoingTransferMsgsScript(strProcessorName);
    String strIncoming = generateIncomingTransferMsgsScript(strProcessorName);

    if (stroutgoing.trim().length() > 0)
      strAgentDefinition = stroutgoing + "\n" + MESS_THEN + "\n";
    strAgentDefinition = strAgentDefinition + strIncoming;

  }
  return "a(" + strProcessorName + ", ID) :: \n" + strAgentDefinition;
}

/**
 * Function to generate and print the LCC interaction model script for multi
 * agents.
 */
public void printDefs(){
  String [] names= objReader.getProcessorNames();
  for (int i=0; i < names.length; i++)
  {
    System.out.println(generateProcessorDefinition(names[i])+"\n");
  }
  //Also print the definition for the system peer.
  System.out.println(generateProcessorDefinition(Reader.PROC_SYSTEM));
}

public static void main(String[] args) {
  Translator translator = new Translator("E:\\FourthTerm\\Thesis\\DemoJunk\\TavToLcc\\examples\\BiomartAn
  translator.printDefs();
}
```

```
}
```

A.5.2 Reader.java

```
/**
 *  Class: Reader
  * Description: This class is a helper class for Translator class. This class
  * is able to load the SCUFL XML file using JDOM. Through its functionalities
  * it is then able to extract the desired workflow information from workflow
  * file and return it to the caller.
 */

import org.jdom.*;
import org.jdom.input.SAXBuilder;
import java.io.FileInputStream;
import java.util.List;
import java.util.ListIterator;
import java.util.Vector;

public class Reader {
  public static final String ELE_PROCESSOR = "processor";
  public static final String ELE_LINK = "link";
  public static final String ELE_COOR = "coordination";
  public static final String ELE_ACTION = "action";
  public static final String ELE_TARGET = "target";
  public static final String ELE_CONDIT = "condition";

  public static final String ATT_NAME = "name";
  public static final String ATT_SINK = "sink";
  public static final String ATT_SOURCE = "source";

  public static final String PROC_SYSTEM = "system";

  private SAXBuilder builder;
  private Document document;

  /**
   * Function to get all the postconditions for the processor.
   * @param strProcName String = SCUFL processor name
   * @return Vector = post condition elements for the processor
   */
  public Vector getPostConditionProcessorNames (String strProcName){
    Vector toRet = new Vector();
    Element ele = document.getRootElement();

    List list_coords = ele.getChildren(ELE_COOR, ele.getNamespace());
    if (!list_coords.isEmpty()){
      ListIterator iter_cord = list_coords.listIterator();
      while (iter_cord.hasNext())
       {
         Element coor = (Element) iter_cord.next();
         //see the condition part
         Element condition = (Element) coor.getChild(ELE_CONDIT, coor.getNamespace());
         Element target = (Element) condition.getChild(ELE_TARGET, condition.getNamespace());
         if (target.getText().equals(strProcName))
         {
           //Get the name of the processor in the action part which needs to be
           //sent a Go message.
           Element action = (Element)coor.getChild(ELE_ACTION, coor.getNamespace());
           Element acttarget = (Element)action.getChild(ELE_TARGET, action.getNamespace());
           toRet.add(acttarget.getText());
```

```
      }
     }
     if (toRet.size()>0)
       return toRet;
     else return null;
  }
  else
    return null;

}

/**
 * Function to fetch all the preconditions for a processor.
 * @param strProcName String = SCUFL processor name
 * @return Vector = pre condition elements for the processor.
 */
public Vector getPreconditionProcessorNames (String strProcName){
  Vector toRet = new Vector();
  Element ele = document.getRootElement();

  List list_coords = ele.getChildren(ELE_COOR, ele.getNamespace());
  if (!list_coords.isEmpty()){
    //check to see if the processor name passed exists in the action part
    //of the coordination. If so return the condition part's processor name.
    ListIterator iter_cord = list_coords.listIterator();
    while (iter_cord.hasNext())
    {
      Element coor = (Element) iter_cord.next();
      //get action part
      Element action = coor.getChild(ELE_ACTION, coor.getNamespace());
      if (action != null){
        //get target element
        Element target = (Element) action.getChild(ELE_TARGET, action.getNamespace());
        if (target != null){
          String strProc = target.getText();
          if (strProc.equals(strProcName))
          {
            //process exist in the target part.
            //get the condition part processor name.
            Element condition = coor.getChild(ELE_CONDIT, coor.getNamespace());
            Element target_cond = condition.getChild(ELE_TARGET, condition.getNamespace());
            toRet.add(target_cond.getText());
          }
        }
      }
    }
    if (toRet.size() > 0)
      return toRet;
    else return null;
  }
  else
    return null;
}

/**
 * Function to get the list of link elements where processor name passed
 * exists as source or sink.
 * @param strProcName String = SCUFL processor name
 * @param AttribName String = (sink or source)
 * @return Vector = list of link elements where processor exists as sink/source
 */
public Vector getLinkWithProcessorAsSinkOrSource(String strProcName, String AttribName){
```

```
  Vector toRet = new Vector();
  String value;
  Element ele = document.getRootElement();

  List list_links = ele.getChildren(ELE_LINK, ele.getNamespace());
  if (!list_links.isEmpty()){
    ListIterator iter = list_links.listIterator();
    while (iter.hasNext())
    {
      Element link = (Element) iter.next();
      value = link.getAttributeValue(AttribName);
      if (value.indexOf(":") != -1) //the string is a normal processor
      {
        String parts[] = value.split(":");
        if (parts[0].equals(strProcName))
          toRet.add(link);
      }
      else; //sink or source belongs to the system.
    }
    return toRet;
  }
  else
    return null;
}

/**
 * Funtion to preprocess the workflow file to accommodate the System processor.
 */
public void PreProcessingForSystemPeer(){
  Vector toRet = new Vector();
  String value;
  Element ele = document.getRootElement();
  List list_links = ele.getChildren(ELE_LINK, ele.getNamespace());
  if (!list_links.isEmpty()){
    ListIterator iter = list_links.listIterator();
    while (iter.hasNext()){
      Element link = (Element) iter.next();
      value = link.getAttributeValue(ATT_SINK);
      if (value.indexOf(":") == -1){
        //system peer -> update attribute value
        link.setAttribute(ATT_SINK, PROC_SYSTEM+":"+value);
      }

      value = link.getAttributeValue(ATT_SOURCE);
      if (value.indexOf(":") == -1){
        //system peer -> update attribute value
        link.setAttribute(ATT_SOURCE, PROC_SYSTEM+":"+value);
      }
    }
  }
}

/**
 * Funtion to get all the existing processors in the SCUFL workflow
 * @return String[] = list of existing processor
 */
public String[] getProcessorNames (){
  String[] toRet;
  Element ele = document.getRootElement();

  List list_procs = ele.getChildren(ELE_PROCESSOR, ele.getNamespace());
```

```
    if (!list_procs.isEmpty())
    {
      toRet = new String[list_procs.size()];
      ListIterator iter = list_procs.listIterator();
      int i = 0;
      while (iter.hasNext())
      {
        Element proc = (Element) iter.next();
        toRet[i] = proc.getAttributeValue(ATT_NAME);
        i++;
      }
    }
    else
      toRet = null;

    return toRet;
  }

  /**
   * Constructor to load the SCUFL workflow file.
   * @param strName String = SCUFL file path.
   */
  public Reader(String strName) {
    try{
      builder = new SAXBuilder();
      document = builder.build(new FileInputStream(strName));
    }
    catch (Exception e)
    {
      System.out.println( " >>> In function Reader: " );
      e.printStackTrace();
    }

  }
}
```