Construction of an Agent-based Framework for Evolutionary Biology: a Progress Report

Yu Pan Phan Huy Tu Enrico Pontelli Tran Cao Son

Department of Computer Science
New Mexico State University
{ypan,tphan,epontell,tson}@cs.nmsu.edu

Abstract. We report on the development of an agent-based system, called ΦLOG, for the specification and execution of phylogenetic inference applications. We detail the implementation of the main components of the system. In the process, we discuss how advanced techniques developed in different research areas such as domain-specific languages, planning, Web services discovery and invocation, and Web services composition can be applied in the building of the ΦLOG system.

1 Introduction

In the biological sciences, data is accumulating much faster than our ability to convert it into meaningful knowledge. For example, the Human Genome Project and related activities have flooded our databases with molecular data. The size of the DNA sequence database maintained by NCBI has surpassed 15 million sequences and keeps growing at a rapid pace. Our modeling tools are woefully inadequate for the task of integrating all that information into the rest of biology, preventing scientists from using these data to draw meaningful biological inferences. Thus, one of the major challenges faced by computer scientists and biologists *together* is the enhancement of information technology suitable for modeling a diversity of biological entities, leading to a greater *understanding* from the influx of data. Instead of allowing the direct expression of high-level concepts natural to a scientific discipline, current development techniques require mastery of programming and access to low level aspects of software development.

The ΦLOG Project: The ΦLOG project at NMSU is aimed at the development of a computational workbench to allow evolutionary biologists to rapidly and independently construct computational analysis processes in phylogenetic inference. Phylogenetic inference involves the study of evolutionary change of traits (genetic or genomic sequences, morphology, physiology, behavior, etc.) in the context of biological entities (genes, genomes, individuals, species, higher taxa, etc.) related to each other by a phylogenetic tree or genealogy depicting the hierarchical relationship of common ancestors.

The overall objective of the ΦLOG framework is to allow biologists to design computational analysis processes by describing them at the same level of abstraction commonly used by biologists to think and communicate—and not in terms of complex low-level programming constructs and communication protocols. The ΦLOG framework automatically translates these high-level descriptions into executable programs—commonly containing appropriately composed sequences of invocations to existing bioinformatics tools (e.g., BLAST, DNAML).

The ΦLOG framework is characterized by two innovative aspects: the use of a *Domain Specific Language (DSL)* as interface to the biologists and the adoption of an agent-based platform for the execution of ΦLOG programs. These aspects are discussed in the next subsections.

The ΦLOG Language: The ΦLOG framework offers biologists a *Domain Specific Language (DSL)* for the description of computational analysis processes in evolutionary biology. The DSL allows biologists to computationally solve a problem by programming solutions *at the same level of abstraction they use for thinking and reasoning*. In the DSL approach, a language is developed to allow users to build software in an application domain by using programming constructs that are natural for the specific domain. A DSL results in programs that are more likely to be correct, easier to write and reason about, and easier to maintain [12, 15, 20]. The ΦLOG DSL has been extensively described in [25]. The language provides:

- High-level data types representing the classes of entities typically encountered in evolutionary biology analysis (e.g., genes, taxon, alignments). The set of types and their properties have been derived as a combination of existing data description languages (e.g., NEXUS [22]) and biological ontologies (e.g., Bio-Ontology [28]).
- High-level operations corresponding to the transformations commonly adopted in computational analyses for evolutionary biology (e.g., sequence alignment, phylogenetic tree construction, sequence similarity search). The operations are described at a high-level; the mapping from high-level operations to concrete computational tools can be either automatically realized by the ΦLOG execution model, or explicitly resolved by the programmer.
- Both declarative as well as imperative control structures to describe execution flow. Declarative control relies on high-level combinators (e.g., functions, quantifiers) while imperative control relies on sequencing, conditional, and iterative constructs.

The ΦLOG Agent Infrastructure: An essential goal behind the development of ΦLOG is to provide biologists with a framework that facilitates discovery and use of the variety of bioinformatics tools and data repositories publicly available. The Web has become a mean for the widespread distribution of a large quantity of analysis tools and data sources, each providing different capabilities, interfaces, data formats and different modalities of operation. Biologists are left with the daunting task of locating the most appropriate tools for each specific analysis task, learning how to use them, dealing with the issues of interoperability (e.g., data format conversions), and interpreting the results. As a result of this state of things, frequently biologists make use of suboptimal tools, are forced to perform time-consuming manual tasks, and, more in general, are limited in the scope of analysis and range of hypothesis they can explore.

ΦLOG relies on an agent infrastructure, where existing bioinformatics tools and data sources are viewed as *bioinformatics services*. Services are formally described; the agent infrastructure makes use of such formal descriptions and of the content of ΦLOG programs to determine the appropriate sequence of service invocations required to accomplish the task described by the biologist. The reasoning component of the agent is employed to select services and compose them, eventually introducing additional services to guarantee interoperability. The rest of this paper describes in detail the structure of such agent infrastructure.

Related Work: Relatively limited effort has been invested in the use of agent-based technology to facilitate the creation of analysis processes and computational biology applications. TAMBIS [11] provides a knowledge base for accessing a set of data sources, and it can map queries expressed in graphical form to sequences of accesses. Some proposals have recently appeared addressing some of the aspects covered by ΦLOG , such as ontologies for computational biology (e.g., BIOML [13] and Bio-Ontology [28]), interoperability initiatives (e.g., the Bioperl Project [6], XOL project [21] and the TAMBIS project [11]), low-level infrastructure for bioinformatics services (e.g., OmniGene [8], BioMOBY [10], and the DAS [24]), and generic bioinformatics computational infrastructures (e.g., BioCorba [9] and BioSoft [14, 16]).

2 System Overview

The overall architecture of our system is illustrated in Figure 1. The execution of ΦLOG programs will be carried out by an agent infrastructure and will develop according to the flow denoted by the arrows in Figure 1. In this framework, bioinformatics tools are viewed as *Web services*; in turn, each agent treats such services as *actions*, and the execution of ΦLOG programs is treated as an instance of the *planning and execution monitoring problem* [19]. Each data source and tool has to be properly described—in terms of capabilities, inputs and outputs—so that the agent can determine when a particular data source or tool can be used to satisfy one of the steps required by the ΦLOG program. This description process is supported by a *bioinformatics ontologies* for the description of the entities involved in this process. ΦLOG programs will be processed by a compiler and translated into an *abstract plan*, that identifies the high-level actions (i.e., analysis steps) required, along with their correct execution order. The abstract plan is processed by a *configuration component*; the output of the configuration component is a situation calculus theory [26] and a ConGolog program [17]. The ConGolog program represents the underlying skeleton of the plan required to perform the computation described in the original ΦLOG program. The action theory describes the actions that can be used in such plan. These actions correspond to the bioinformatics services that can be employed to carry out the tasks described by the high-level actions present in the abstract plan. The descriptions of such actions are retrieved from a *service broker*, which maintains (DAML-S) descriptions of all registered bioinformatics services.

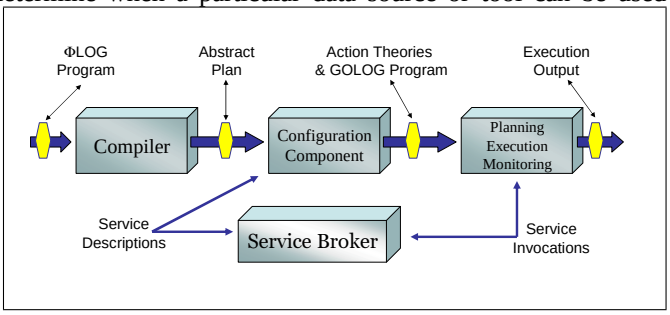

Fig. 1: Overall System Organization

The situation calculus theory and the ConGolog program are then processed by a *planner*; the task of the planner is to develop a *concrete plan*, which indicates how to compose individual bioinformatics services to accomplish the objectives described by the

ΦLOG programs. In the concrete plan, the high-level actions are replaced by invocation calls to concrete bioinformatics services; it might also include additional steps not indicated in the original ΦLOG program, required to support interoperation between services (e.g., data format conversions) and to resolve ambiguities (e.g., tests to select one of possible services). The creation of the concrete plan relies on the technology for *reasoning about actions and change*. The planner is integrated with an execution monitor, which is in charge of executing the concrete plan by repeatedly contacting the broker to request execution of specific services. The execution monitor interacts with the planner to resolve situations where a plan fails and replanning is required.

3 Service Description and Management

Bioinformatics services are described in our framework using DAML-S 0.7, a language built on top of the DAML+OIL ontology for Web Services. We adopt DAML-S over previously developed Web Service languages (e.g., WSDL or SOAP[1]), for its expressiveness and declarativeness. Furthermore, DAML-S has been designed to make Web Services computer-interpretable, thus allowing the development of agents for service discovery, invocation, and composition. As such, it is an ideal representation language for describing bioinformatics services.

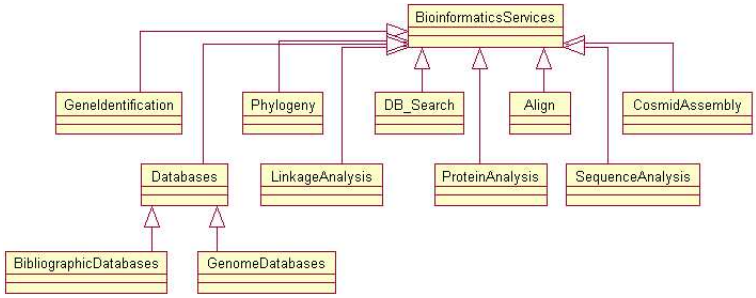

Fig. 2. Part of Service Hierarchy

Service Description: Bioinformatics services in ΦLOG are classified according to a type hierarchy. This classification facilitates the matching between the high-level actions present in a ΦLOG program and the actual services. More details related to this topic will be discussed in Section 5. A part of the service hierarchy is shown in Figures 2. The top class in this hierarchy is called `BioinformaticsServices` and is specified by the following XML element:

```
<daml:Class rdf:ID="BiologyServices">
  <rdfs:label>Biology Service</rdfs:label>
  <rdfs:comment> ... </rdfs:comment>
  <rdfs:subClassOf rdf:resource= "http://www.daml.org/services/
      daml-s/0.7/ProfileHierarchy.daml#Information_Service" />
</daml:Class>
```

All the other classes are derived directly or indirectly from this class. As an example

[1] `http://www.w3.org/TR/wsdl` and `http://ws.apache.org/soap/`

(Figure 2), `BibliographicDatabases` and `GenomeDatabases` are subclasses of the `Databases` class, which, in turn, is a subclass of `BioinformaticsServices`. Both of them describe database-related services which allow users to access different databases. This class includes services such as GDB [3] (Human genomic information), OMIM [4] (Catalogue of human genetic disorders), and EMBASE [7] (Excerpta Medica Database). Their representation is as follows.

```
<daml:Class rdf:ID="GenomeDatabases">
    <rdfs:subClassOf rdf:resource="#Databases"/>
</daml:Class>
<daml:Class rdf:ID="BiliographicDatabases">
    <rdfs:subClassOf rdf:resource="#Databases"/>
</daml:Class>
```

Information about the service classification is stored in the file *datatypes.daml*.[2] In addition to the classification of services, this file contains information about the types of biological entities that are important for the development of our system. As with services, these objects are also organized as a class hierarchy to facilitate reasoning about types of objects. For example, biological sequences are represented as objects of the `Sequences` class. The file *datatypes.daml* also includes some predefined instances of classes. Fig. 3 shows part of these hierarchies.

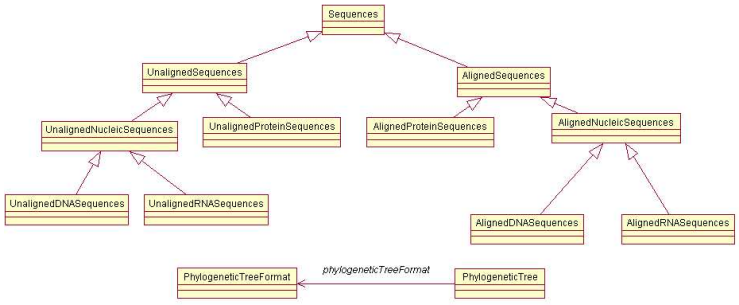

Fig. 3. Part of Biological Object Classification

These hierarchies help us to reason about the services needed to execute a ΦLOG program, including reasoning about the types and the formats of service parameters.

Let us now describe the DAML-S representation of services through an example—the representation of the `ClustalW` service, a multiple sequence alignment program [2]. In DAML-S, each service is characterized by a *profile*, representing the capabilities and parameters of the service, a *process model*, illustrating the workflow of the service, and a *grounding* file, specifying in details how to access the service. The DAML-S representation of the `ClustalW` service is composed of four files[3] described below.

The first file *clustalw-service.daml*[4] stores information about the locations of the profile, the process model, and the grounding:

```
<service:Service rdf:ID="Service_ClustalW">
```

[2] `http://www.cs.nmsu.edu/~tphan/philog/nondet/datatypes.daml`
[3] The complete description can be found at `www.cs.nmsu.edu/~tphan/philog`
[4] `www.cs.nmsu.edu/~tphan/philog/nondet/clustalw-service.daml`

```
  <service:presents rdf:resource="&clw_profile;#Profile_ClustalW"/>
  <service:describedBy rdf:resource="&clw_process;#Process_ClustalW"/>
  <service:supports rdf:resource="&clw_grounding;#Grounding_ClustalW"/>
</service:Service>
```

The second file, *clustalw-profile.daml*[5], is the profile for the `ClustalW` service. It defines the parameters needed for the invocation of this service and specifies the membership of this service in the service classification hierarchy. For example, `ClustalW` is specified as an instance of the class `Align`:

```
<ftypes:Align rdf:ID="Profile_ClustalW">  ...  </ftypes:Align>
```

This file also contains input, output, and precondition elements defining the service's inputs, outputs, and preconditions, respectively. The type of each parameter (input or output) is specified by the `restrictedTo` property, making use of the above biological object classification.

The third file, the process model, provides the necessary information for an agent to use the service, including specifying whether it is an atomic process or a composed process or what are its inputs, outputs, and preconditions. For example, the `ClustalW` service is specified as an atomic process in its process model[6]:

```
<daml:Class rdf:ID="ClustalWProcess">
  <rdfs:subClassOf rdf:resource="&process;#AtomicProcess" />
</daml:Class>
```

Similarly to the service profile, the process model also makes use of the above biological classification to define its input and output parameters.

Finally the grounding model for `ClustalW` specifies the details of how to access the service—details having mainly to do with protocol and message formats, serialization, transport, and addressing. It consists of two complementary parts (i) a DAML-S file specifying the mapping between DAML processes/types and WSDL operations/messages, and (ii) a WSDL file designating the binding of messages to various protocols and formats. The first file is called *clustalw-grounding.daml* and the second file is called *clustalw-grounding.wsdl*. Both of them can be found at `http://www.cs.nmsu.edu/~tphan/philog/`.

Service Management: The services, together with their classification, are registered with the service broker, which is responsible for providing service descriptions to the configuration module and fulfilling service execution requests from the execution monitoring module. We employ the OAA system [29] in the development of the service broker. To facilitate these tasks, a *lookup agent* and several *service wrappers* have been developed. The lookup agent receives high-level action names from the compiler and will match these actions with possible available services. For example, a request for a high-level action `align` will be answered with the set of all available alignment services, such as `service_clustalw` and `service_dialign`. This process will be detailed in Section 5. Service wrappers have been developed for the purpose of executing the services since most of the bioinformatics services are still offered through HTTP-requests and not as Web services. Agents—playing the role of service wrappers—are ready for the instantiation and execution of bioinformatics services.

[5] `www.cs.nmsu.edu/~tphan/philog/nondet/clustalw-profile.daml`
[6] `www.cs.nmsu.edu/~tphan/philog/nondet/clustalw-process.daml`

4 ΦLOG Compiler

The objective of the ΦLOG compiler is to process a program written in ΦLOG and produce as output a high-level sketch of the execution plan (the *abstract plan*) and a symbol table, describing the entities involved in the computation, in terms of their names and types. The main tasks of the ΦLOG compiler include: syntax analysis, type checking, and construction of the abstract plan.

Syntax Analysis: Each ΦLOG program contains a sequence of declarations and a sequence of statements. The declaration part is used to:

- Describe the data items (variables) used by the program;
- Allow users to select the computational components to be used during execution—e.g., associate high-level ΦLOG operations to specific bioinformatics tools;
- Provide parameters affecting the behavior of the different components.

Each data item used in the program must be properly declared. Declarations are of the type `<variable> : <type> [<properties>]` and are used to explicitly describe data items, by providing a name (`<variable>`), a description of the nature of the values that are going to be stored in it (`<type>`) and properties of the item. For example, `gene1 : Gene ( gi | 557882 )` declares an entity called `gene1`, of type `Gene`, and identifies the initial value for this object—the gene with accession number `557882` in the GenBank database.

Declarations are also used to identify computational components to be used during the execution—this allows the user to customize some of the operations performed. For example, a declaration of the type

```
align_sequences : Operation(CLUSTALW -- alignment = full,
    score type = percent, matrix = pam, pairgap = 4 );
```

allows the user to configure the language operation `align_sequences`—a ΦLOG operation to perform sequence alignments—by associating this operation with the ClustalW alignment program, with the given values for the input parameters.

Variable assignments are expressed as: `<output variable> is <operation>(<input variable>).` In this prototype, we focused on a subset of the possible classes of operations—i.e., `<searchOp>`, `<alignOp>`, `<buildTreeOp>`, and `<specificOp>`.

Type Checking: All variables used in a ΦLOG program must be declared with specific types. ΦLOG provides two classes of datatypes. The first class includes generic (non-domain specific) datatypes, while the second class includes all those datatypes that are domain-specific, like DNA Sequence, Protein, etc. These domain-specific types are defined in our type system (see Fig. 3). There are two major types of type checking

- Type checking against attributes of objects;
- Type checking against input and output variables of operations.

Domain specific datatypes contain attributes that are specific to each type. Consider the following ΦLOG program segment

```
g1 : Gene ( gi | 557882 )
se : Sequence
se is sequence(g1)
```

It assigns to the variable `g1` the Gene having accession number `GI | 557882` and extracts its sequence data, which is stored in the variable `se`. The compiler must check

the datatype hierarchy to verify that `GI|557882` is a legal value for an object of type `Gene`—i.e., it is a well-formed accession number—and an attribute called `sequence` with type `Sequence` exists for the type `Gene`. Attribute and type mismatches will cause compiling error. Type checking is also performed for each operation in the program. Datatypes of input and output variables are defined in our services ontology (see Figure 2). The ΦLOG compiler must check the validity of such parameters; e.g., `s2 is align(s1)` performs a multi-sequence alignment operation on the data item `s1`, storing the result in data item `s2`. To be able to perform the action, `s1` must be of type `UnalignedSequences` (i.e., a set of unaligned sequences) and `s2` must be of type `AlignedSequences`.

Operations Identification and Abstract Plan Assembly: As described in the syntax analysis section, the current preliminary prototype focuses on a limited classes of operations (explored for feasibility purposes):

```
  <operation> ::= <searchOp> | <alignOp> | <buildTreeOp> | <specificOp>
  <searchOp> ::=  <variable> : <variable> is <complexType> and
                  <attribute>(<variable>) <verb> <literal>
  <alignOp> ::= align
  <buildTreeOp> ::= build_tree
```

Database search operations are conveniently expressed using intensional sets. E.g.,

```
p is { x : x is Gene and name(x) contains "fever" }
```

searches a nucleotide database—automatically inferred from the type of the collected variable `x`—for all genes whose name contains the keyword `''fever''`, and the resulting collection of genes is stored in the variable `p`.

Each syntactic occurrence of an operations leads to the generation of one high-level action in the abstract plan assembled by the compiler. The identification of the operation is accomplished by navigating the services hierarchy, with the goal of locating the most specific class of services corresponding to the specified operation. The operation provides a link to the most general class of services in the ontology corresponding to such operation (e.g., the `align` operation used in a ΦLOG program will link to the general class of sequence alignment services in service hierarchy); the usage of the operation—and, in particular, the type of the parameters, inputs, and outputs—will constraint the focus on appropriate subclasses of services.

The ΦLOG language allows us also to directly refer to specific services (e.g., either through a declaration, as illustrated in the previous section, or directly as an operation). For example, `s is ClustalW_JP(p)` identifies the `ClustalW` multi-sequence alignment service located at `clustalw.genome.ad.jp`. This operation is described in the service hierarchy, with input type `UnalignedSequences` and output type `AlignedSequences`. However, use of specific service is not recommended in a ΦLOG program because user then can not take use of the power of dynamic service plan composition of the ΦLOG framework.

As another example, the service hierarchy offers three subclasses of `build_tree` operation—used to construct a phylogenetic inference tree—that use different algorithms: `ParsimonyAlign`, `DistanceMatrixAlign`, and `MaximumLikelyhoodAlign`. These operations are differentiated by their input parameters and the

ΦLOG compiler must be able to find the correct match. For example,

```
p : UnalignedSequences
m : DistanceMatrix
s is align(p, m)
```

identifies the operation `DistanceMatrixAlign(p, m)` because it has two inputs: a set of unaligned sequences and a distance matrix.

The output produced by the compiler is an abstract plan. The abstract plan is a ConGolog program whose actions are high-level actions. Each service is described by a three elements tuple: $\langle A, IL, OL\rangle$ where A is the operation name, IL is the list of A's input parameters and OL is the list of A's output parameters respectively. Each input or output is of the form $(name, type, value)$, where $name$, $type$ and value are the name, type and value of the input/output respectively. The value of an input or output must be either a constant or a variable.

In addition to the abstract plan, the output of the compiler also contains information about all the variables used in the ΦLOG program and a list of high-level actions. Specifically, for each variable X of type T in the program, there is a corresponding fact $var(X, T)$ in the output. As an example, consider the ΦLOG program fragment:

```
p : UnalignedSequences;
s : AlignedSequences;
t : PhylogeneticTree;
p is  x : x is Gene and name(x) contains "fever";
s is align(p);
t is build_tree(s);
```

This simple program defines a sequence of operations—first search a database finding all the genes contains the keyword "`fever`", then conduct a multiple sequence alignment operation on the returned sequence set, and finally build a evolution tree based on the aligned sequence set. The output of the compiler is a list of three high-level actions `db_search, align`, and `build_tree` and the Prolog program

```
plan([(db_search, [(db,str,nucleotide),(term,str,fever)],
            [(sequence,unalignedsequences,p)]),
      (align, [(sequence,unalignedsequences,p)],
            [(sequence,alignedsequences,s)]),
      (build_tree,[(inFile,alignedsequences,s)],
            [(outputFile,phylogenetictree,t)])]).
var(t,phylogenetictree). var(s,alignedsequences).
var(p,unalignedsequences).
```

The fact `plan(...)` represents the ΦLOG program and the set of facts of the form `var(.,.)` list the variables used in the program.

5 Configuration Component

The configuration component plays an important role in preparing the ΦLOG program for execution. Its input is an abstract plan from the compiler. Its output is a ConGolog program with an underlying situation calculus theory, that will be used

by the Planning and Execution monitoring module to execute the ΦLOG program. For the background behind this design and its advantages, we refer the reader to [27]. Figure 4 shows the phases of the configuration component. We next describe these phases in more detail.

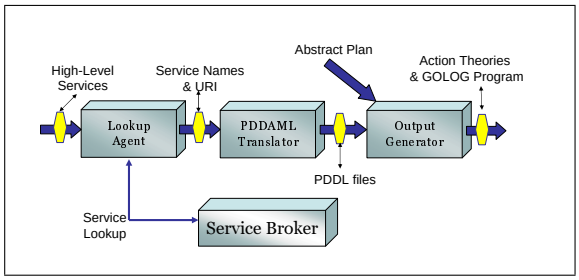

Fig. 4: Configuration Component

5.1 DAML-PDDL translator

The DAML-PDDL translator, in concert with the services broker (which maintains the service registry), is responsible for collecting of DAML-S service descriptions needed for the execution of the ΦLOG program and converting them into PDDL files. The lookup agent, after receiving the list of high-level actions from the compiler, will request the broker for a list of bioinformatics services which can be used to realize the high-level actions. This list of services, which contains information about service names and their locations (URIs), is then forwarded to the translator. For example, the `db_search` service is realized by the bioinformatics services `ncbi` and `blast` at `http://www.cs.nmsu.edu/~tphan/philog/`. For each service, the translator will download the service descriptions from the specified URIs and convert them to PDDL files.[7]

The DAML-PDDL translator used in this project, called PDDAML, is an automatic translator between PDDL and DAML from [5]. It is worth noticing that this step could be eliminated and replaced by a module that translates DAML-S service descriptions directly into a situation calculus theory. However, we still adopt this path for several reasons. First of all, the language DAML-S is still under development, and any changes in its specification would also mean changes to our system. Secondly, the language PDDL is well-known and accepted as the input language for many planning systems. Furthermore, the DAML-S parser and analyzer are being developed and updated by the DAML coalition. By using PDDAML, we make our system less sensitive to changes in the DAML-S specification and avoid the need of writing programs for processing DAML-S specifications.

Each DAML-S file (service, profile, process model, or grounding)—as described in Section 3—is translated into a PDDL file, often referred to as a *PDDL domain*. Each PDDL domain consists of several sections specifying the external domains that are extended by the current domain and defining the domain's entities and their relationships such as data types, objects, predicates, axioms, etc. E.g., the PDDL domain representing the profile of the service `ClustaW`,[8] named `clustalw-profile-ont`, uses the external domains `clustalw-service-ont` (representing the service) and `clustalw-process-ont` (representing the process model) and defines objects named `Profile_ClustalW`, `Sequences`, `OutputSequences`, etc.; it also contains axioms describing the input, output, and precondition of the services.

[7] More precisely, the output is in WebPDDL format.

[8] Space limitation does not allow us to display the output of the translator here. Readers interested in the details can find it at `http://www.cs.nmsu.edu/~tphan/philog/`.

5.2 Generating the Situation Calculus Theory and the ConGolog Program

In the second phase, the configuration component takes the output from the DAML-PDDL translator (a collection of PDDL files) and from the compiler (the abstract plan) and generates the situation calculus theory and the ConGolog program for the Planning and Execution module. This is done in two steps. First, the set of PDDL domains is combined into a single Prolog file whose facts and rules represent the objects and axioms in the PDDL files. To avoid naming conflicts between entities from different domains, we associate to each domain a unique string, called *tag*, and prefix each entity of the domain with the corresponding tag. Consider, for example, the object `Sequences`, that represents the input of `ClustalW`, and is defined in the PDDL domain `clustalw-profile-ont` (originated from *clustalw-profile.daml*) with the type `Unaligned-Sequences`. Assume that this domain is associated with the tag `F17`. The object is translated into a predicate `unalignedSequences(F17_Sequences)` of the Prolog program.

The final step in the configuration component is to generate the situation calculus theory and to formulate the ConGolog program corresponding to the ΦLOG program. This process involves collecting all the necessary information about a particular service from the Prolog program produced in the previous step and from the abstract plan—the output of the compiler (*see* Section 4). This step is performed as follows.

Generating the Facts. Each variable X of type T in the ΦLOG program corresponds to a fact $T(X)$ in the action theory. Similarly, a constant C of type T has the corresponding fact $T(C)$. For example, for the output of the ΦLOG program described in Section 4, the destination theory contains the following facts:

```
phylogenetictree(p).   alignedsequences(s).   unalignedsequences(t).
str(nucleotide).       str(fever).
```

where `p`, `s` and `t` are variables while `nucleotide` and `fever` are constants.

Generating the Fluents. For each variable X used in the ΦLOG program, there is a corresponding fluent `variable(X)` in the destination ConGolog program. In addition, there is one more fluent `has_value(X)` to indicate whether that variable has been assigned some value or not. Initially, no variable has been assigned a value.

```
prim_fluent(variable(p)).    prim_fluent(variable(s)).
prim_fluent(variable(t)).
prim_fluent(has_value(X)) :- prim_fluent(variable(X)).
```

Besides, it might be the case in which an input of an action is required to have some fixed value. For example, the abstract plan in Section 4 requires that all the `db-search` services have *"nucleotide"* as the value of their first argument and *"fever"* as the value of their second argument. To deal with this case, we use a fluent of the form $value(X, V)$ to say that the value of the input X must be V. The meaning of this kind of fluents will become more precise when we discuss the executability condition of an action in the following parts. E.g., the translator will automatically generate the following fluents for the ΦLOG program output above.

```
prim_fluent(value(f13_db,nucleotide)).   prim_fluent(value(f13_term,fever)).
prim_fluent(value(f0_db,nucleotide)).    prim_fluent(value(f0_term,fever)).
```

Furthermore, depending on the service description, the situation calculus might have some additional fluents. E.g., since the precondition of `ClustalW` involves the format

property, the theory will contain the fluent `format(X,V)`, indicating that the format of object `X` is `V`.

Generating the Actions. Each service occurring in the previous step corresponds to an action in the destination theory, whose parameters are the inputs and outputs of the service. The translator will automatically assign a unique variable name for each input and output of a service. E.g., the service ClustalW corresponds to the following action in the action theory:

```
prim_action(service_clustalw(input(F17_sequences),output(F17_outputsequences))) :-
                                        unalignedsequences(F17_sequences),
                                        alignedsequences(F17_outputsequences).
```

It says that the service ClustalW has an input `F17_sequences` and an output `F17_output sequences`, where `F17_sequences` and `F17_outputsequences` are of the types `unalignedsequences` and `alignedsequences` respectively.

In several cases, some services in the local database might be used to formulate actions in the theory. For instance, we notice that we may need to do some kind of format conversions for our ΦLOG program. Hence, all the format conversion services in the local database are looked up and included in the theory. In the future, the search for related services will be done online, through the service broker.

Generating the Executability Conditions. The following is an example of the executability condition for the ClustalW service.

```
executable(service_clustalw(input(F17_sequences),output(F17_outputsequences)),
        and(format(F17_sequences,sf_ncbi),or(value(f17_sequences,F17_sequences),
                and(variable(F17_sequences),has_value(F17_sequences))))):-
        unalignedsequences(F17_sequences),alignedsequences(F17_outputsequences).
```

The intuition behind the above condition is that, for the service ClustalW to be executable, it requires each of its input parameters either to be a variable that is already assigned to some value or to have some default value. In addition, it also requires that the format of the input `F17_sequences` is `sf_ncbi`.

Generating Effects. One type of effect of an action is that its outputs will be assigned some value. E.g., the effect of the ClustalW service in the action theory looks like:

```
causes_val(service_clustalw(input(F17_sequences),output(F17_outputsequences)),
                has_value(F17_outputsequences),true,true) :-
        unalignedsequences(F17_sequences), alignedsequences(F17_outputsequences).
```

The other type of effect relates to effects that are explicitly described in the service description. For example, the BLAST search service has an effect stating that the format of its output is `sf_blast`. This is represented as follows.

```
causes_val(service_blast(input(F13_db,F13_term),output(F13_outputsequences)),
                        format(F13_outputsequences,sf_blast),true,true) :-
        str(F13_db),str(F13_term),unalignedsequences(F13_outputsequences).
```

Generating the Initial State. As mentioned, for the ΦLOG program we are considering, initially no variable has been assigned any value. The ConGolog encoding is:

```
initially(variable(p),true). initially(variable(s),true).
initially(variable(t),true). initially(has_value(t),false).
initially(has_value(s),false). initially(has_value(p),false).
initially(value(f13_db,nucleotide),true).
initially(value(f13_term,nucleotide),true).
```

initially(value(f0_db,nucleotide),true).
initially(value(f0_term,fever),true).

Generating ConGolog Programs. Based on the abstract plan and the domain description, a ConGolog program representing the concrete plan can be constructed. The following is an example of the ConGolog program for the ΦLOG program in Sect. 4.

```
proc(plan,[service_ncbi(input(F0_db,F0_term),output(F0_outputsequences))
    make_doable
    service_dialign(input(F21_sequence),output(F21_outputsequences)):
    service_clustalw(input(F17_sequence),output(F17_outputsequences))
    make_doable
    service_treeview(input(F29_inputfile),output(F29_outputphylogenetictree)):
    service_dnaml(input(F25_inputfile),output(F25_outputphylogenetictree))]).
```

Notice that any pair of consecutive plan steps has a construct `make_doable` in-between. This construct, introduced in [23], is a relaxation of ConGolog's sequence construct.

6 Planning and Execution Monitoring Module

The input of the planning and execution monitoring module consists of a ConGolog program and a situation calculus theory which represents the original ΦLOG program and the bioinformatics services, respectively. The module's job is to execute the ConGolog program. To do so, it repeatedly generates traces of the ConGolog program and then executes them until at least one concrete plan succeeds, or all of them fail (Fig. 5).

6.1 Planning

The main job of this component is to find a possible trace of the ConGolog program which can be successfully executed and then executes such a trace. Given a ConGolog program and the underlying situation calculus theory, this problem can be solved in different ways by employing different ConGolog interpreters [17, 19]. In this paper, we use an off-line ConGolog interpreter with the insertion constructor 'make_doable' from [23] to generate traces, which we will call hereafter *concrete plans*.

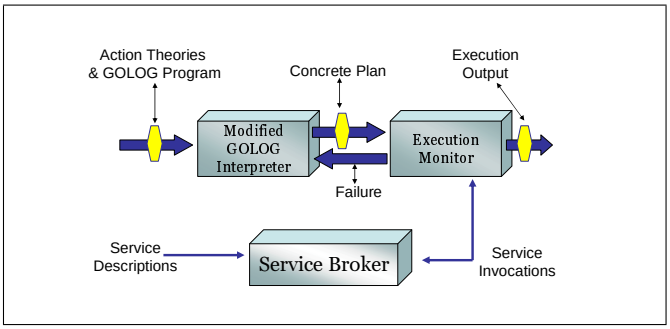

Fig. 5: Planning and Execution Monitoring Module

We prefer the off-line interpreter over the on-line interpreter for different reasons. First of all, the effects of the actions in our ConGolog programs do not change over time, i.e., the execution of a service with the same set of input will yield the same output regardless of its execution time. In this sense, domains in our application satisfy the IPR condition of [23], and therefore this model of planning and execution monitoring is suitable. In addition, there are some services whose runtime is large. As such, a service should be invoked only if it can lead to a successful execution of the program at hand.

This property cannot be satisfied by an on-line interpreter, since it does not guarantee completeness [19].
The use of the insertion constructor allows ΦLOG 's users to write ΦLOG programs without the need of worrying about the data conversion operator in their programs. This simplifies the process of writing ΦLOG programs considerably since the number of data formats currently used by bioinformatics services is huge, and each service only works with certain formats. During the planning phase, the interpreter will automatically insert the *data format conversion* operators into the program, whenever needed. Due to the frequent use of the format conversion utility, we decided to add the situation calculus representation of the format conversion service to every situation calculus theory generated by the configuration component.
To illustrate this process, consider the ConGolog program and the corresponding situation calculus theory from the last section. A possible trace of this program is:

```
| ?- do(plan,s0,S).
S = do(service_treeview(input(s),output(t)),
    do(service_clustalw(input(p),output(s)),
    do(service_ncbi(input(nucleotide,fever),output(p)),s0)))) ?
```

Suppose that the output format of the service NCBI does not match the input format of the service ClustalW. In this case, the output of the planning process is

```
| ?- do(plan,s0,S).
S = do(service_treeview(input(s),output(t)),
    do(service_clustalw(input(p),output(s)),
    do(conversion(input(p),output(p)),
    do(service_blast(input(nucleotide,fever),output(p)),s0)))) ?
```

The action `conversion(input(p),output(p))`, that converts the output format of `service_blast` into a format suitable to `service_clustalw`, is the difference between the traces. It ensures that the sequence of actions is executable from `s0`.

In order to deal with conditional and loop statements in ΦLOG programs we have modified the ConGolog interpreter and its output so that it can deal with conditions whose truth value can only be determined at runtime. We choose to do so instead of using one of the available modified ConGolog interpreters, such as IndiGolog [18], for the same reasons that make us favor an off-line over an on-line ConGolog interpreter. Presently, whenever the interpreter cannot evaluate a condition in a conditional/loop statement, the planning process will continue with the guess that the condition is true/false, thus leaving the job of evaluating the condition for the execution monitoring module. If the evaluation of the condition turns out to be not different than the guess, the execution monitoring module will report a failure (i.e., a backtrack occurs) and the planning process will continue with the opposite guess that the condition is false/true, respectively. To illustrate this, let us consider the ConGolog program $s_1;\ \mathbf{if} v = 2\ \mathbf{then}\ s_2\ \mathbf{else}\ s_3$, which involves three services s_1, s_2, s_3 where s_1 computes the value of a parameter v, $0 \leq v \leq 3$. The off-line ConGolog interpreter will fail to find a trace of this program since it cannot evaluate the condition $v = 2$ if the service s_1 has not been executed. In our interpreter, the first output is $s_1; (v = 2)?; s_2$ (obtained by guessing that $v = 2$ is true). If a backtrack occurs, the next output is $s_1; (\neg(v = 2))?; s_3$.

6.2 Execution Monitoring

The result of the planning process is a concrete plan which is a sequence of bioinformatics services and test conditions. The execution monitoring component will execute the concrete plan by sequentially executing each services or test for the correctness of the condition of the plan. If the service fails or the condition is not satisfied, then the plan execution fails.

It should be noted that if the low-level services occurring in the concrete plan are web services, i.e., they are properly constructed and described using a web service markup language (DAML and WSDL in our case), the invocation of the service is just a matter of using a standard parser to parse the service grounding information and construct invocation messages accordingly. In the current prototype we have created simple agent wrappers for the services to support service invocation. Each wrapper agent must register their functionalities with the OAA broker—in this case, the functionalities provide the name of the service and the invocation parameters. E.g.,

```
oaa_Register(parent, 'ClustalW_JP',
     [clustalw_jp([(sequence, _Sequence)], _Resp)], [])
```

registers with OAA a service called 'ClustalW_JP' which takes one input parameter named 'sequence'. The service invocation is simply a request to the OAA broker for execution of one particular service:

```
oaa_Solve(clustalw_jp([(sequence, Sequence)], Result), [])
```

The wrapper agent will handle the actual service invocation—i.e., building the connection between client and server, constructing the message using either HTTP GET or POST method, parsing the returning message, and storing the result.

In case of execution failure—e.g., a time-out or loss of connection to the remote provider—the monitor will take appropriate actions. Repair may involve either repeating the execution of the service or re-entering the configuration agent. The latter case may lead to exploring alternative ways of instantiating the partial plan, to avoid the failing service. The replanning process is developed in such a way to attempt to reuse as much as possible the part of the concrete plan executed before the failure.

7 Conclusions and Future Work

This paper reports the work that has been done so far in our ΦLOG project. It demonstrates the feasibility of applying agent technologies in phylogenetic inference applications. The main achievement in this phase is the development of the ΦLOG compiler, the configuration component, the execution monitor, and the integration of these components within the OAA system and the ConGolog interpreter. The current system can be used to work with a small class of ΦLOG programs. Much work is still needed before we can get a system that can execute ΦLOG programs as described in [25], i.e., most general ΦLOG programs. This will be our concentration in the near future. Among others, we plan to complete the compiler and the configuration component to allow for control constructors in ΦLOG programs. This will also demand changes in the planning and execution monitoring module. We would also like to improve the planning and execution monitoring module so that results that have been computed by a failed concrete plan can be reused as much as possible in the replanning process.

References

1. Entrez, The Life Sciences Search Engine. www.ncbi.nlm.nih.gov/Entrez.
2. European Bioinformatics Institute. http://www.ebi.ac.uk/clustalw/.
3. Gene Data Bank. http://gdbwww.gdb.org/.
4. OMIM, Online Mendelian Inheritance in Man. www.ncbi.nlm.nih.gov/omim.
5. PDDAML – An Automatic Translator Between PDDL and DAML. http://www.cs.yale.edu/homes/dvm/daml/pddl_daml_translator1.html.
6. The Bioperl Project. www.biperl.org.
7. Human Genome Mapping Project Resource Center. www.hgmp.mrc.ac.uk/MANUAL/.
8. OmniGene: Standardizing Biological Data Interchange Through Web Services, omnigene.sourceforge.net, 2001.
9. BioCORBA Project. www.biocorba.org, 2002.
10. The BioMOBY Project. biomoby.org, 2002.
11. P.G. Baker et al. TAMBIS: Transparent Access to Multiple Bioinformatics Information Sources. In *Int. Conf. on Intelligent Systems for Molecular Biology*, 1998.
12. T. Ball, editor. *Proc. of the 2nd Conference on Domain-specific Languages*. ACM Press, 2000.
13. R. Beavis. The Biopolymer Markup Language (BIOML). TR, ProteoMetrics, LLC, 1999.
14. S. Cao et al. Application of Gene Ontology in Bio-data Warehouse. In *6th Annual Bio-Ontologies Meeting*. 2003.
15. W. Codenie, K. De Hondt, P. Stayaert, and A. Vercammen. From custom applications to domain-specific frameworks. *Communications of the ACM*, 40(10):70–77, 1997.
16. F. Corradini, L. Mariani, and E. Merelli. A Programming Environment for Global Activity-based Aplications. In *WOA, Workshop on Agents*, 2003.
17. G. De Giacomo, Y. Lespérance, and H. Levesque. *ConGolog*, a concurrent programming language based on the situation calculus. *Artificial Intelligence*, 121(1-2):109–169, 2000.
18. G. De Giacomo, H. J. Levesque, and S. Sardiña. Incremental execution of guarded theories. *ACM Transactions on Computational Logic*, 2(4):495–525, 2001.
19. G. De Giacomo, R. Reiter, and M. Soutchanski. Execution monitoring of high-level robot programs. In *KRR'98*, pages 453–465. Morgan Kaufmann Publishers, 1998.
20. G. Gupta and E. Pontelli. Specification, Implementation, and Verification of Domain Specific Languages: a Logic Programming-based Approach. In *CL: from LP into the Future*. Springer, 2001.
21. A.H. Karp. Programming for Parallelism. *Computer*, 20, May 1987.
22. D. R. Maddison, D.L. Swofford, and W.P. Maddison. NEXUS: An Extensible File Format for Systematic Information. *Syst. Biol.*, 464(4):590–621, 1997.
23. S. McIlraith and T.C. Son. Adapting golog for composition of semantic web services. In *(KR'2002)*, pages 482–493. Morgan Kaufmann Publisher, 2002.
24. S. Pearson. DAS: Open Source System for Exchanging Annotations of Genomic Sequence Data. Technical report, Open Bioinformatics Foundation, 2002.
25. E. Pontelli et al. Design and Implementation of a Domain Specific Language for Phylogenetic Inference. *J. of Bioinformatics and Computational Biology*, 2(1):201–230, 2003.
26. R. Reiter. *KNOWLEDGE IN ACTION: Logical Foundations for Describing and Implementing Dynamical Systems*. MIT Press, 2001.
27. T.C. Son et al. An Agent-based Domain Specific Framework for Rapid Prototyping of Applications in Evolutionary Biology. In *1st Workshop on Declarative Agent Languages and Technologies*, 2003.
28. R. Stevens. Bio-Ontology Reference Collection, cs.man.ac.uk/~stevens/onto-publications.html.
29. R. Waldinger. Deductive composition of Web software agents. In *Proc. NASA Wkshp on Formal Approaches to Agent-Based Systems, LNCS*. Springer-Verlag, 2000.