An Agent-based Domain Specific Framework for Rapid Prototyping of Applications in Evolutionary Biology

T.C. Son[1], E. Pontelli[1], D. Ranjan[1], B. Milligan[2], and G. Gupta[3]

[1] Department of Computer Science
New Mexico State University
{tson,epontell,dranjan}@cs.nmsu.edu
[2] Department of Biology
New Mexico State University
brook@biology.nmsu.edu
[3] Department of Computer Science
University of Texas at Dallas
gupta@utdallas.edu

Abstract. In this paper we present a brief overview of the ΦLOG project, aimed at the development of a domain specific framework for the rapid prototyping of applications in evolutionary biology. This includes the development of a domain specific language, called ΦLOG, and an agent-based implementation for the monitoring and execution of ΦLOG's programs. A ΦLOG program—representing an intended application from an evolutionary biologist—is a specification of *what to do* to achieve her/his goal. The execution and monitoring component of our system will automatically figure out *how to do* it. We achieve that by viewing the available bioinformatic tools and data repositories as *web services* and casting the problem of execution of a sequence of bioinformatic services (possibly with loops, branches, and conditionals, specified by biologists) as the web services composition problem.

1 Introduction and Motivation

In many fields of science, data is accumulating much faster than our ability to convert it into meaningful knowledge. This is perhaps nowhere more true than in the *biological sciences* where the Human Genome Project and related activities have flooded our databases with molecular data. The size of the DNA sequence database (e.g., at NCBI), for example, has surpassed 15 million sequences and 17 billion nucleotides, and is growing rapidly. Our modeling tools are woefully inadequate for the task of integrating all that information into the rest of biology, preventing scientists to effectively take advantage of these data in drawing 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 relationships and processes, 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 software development techniques require mastery of computer science and access to very low level aspects of software development in order to construct significantly

complex applications. Even in places where attempts to introduce domain-specific concepts have been made—e.g., design of database formats—scientists are hampered in their efforts by complex issues of interoperation. As a result, currently only biologists with strong quantitative skills and high computer literacy can realistically be expected to undertake the task of transforming the massive amounts of available data into real knowledge. Very few scientists (domain experts) have such computing skills; even if they do, their skills are better utilized in dealing with high-level scientific models than low-level programming issues. To enable scientists to effectively use computers, we need a well-developed methodology, that allows a domain expert (e.g., a biologist) to solve a problem on a computer by developing and programming solutions at the same level of abstraction they are used to think and reason, thus moving the task of programming from software professionals to the domain experts, the end-users of information technology. This approach to software engineering is commonly referred to as *Domain Specific Languages* and it has been advocated by many researchers over the years [29]. The relevance of domain-specific approaches to bioinformatics has been underlined by many recent proposals (both in computer science as well as in biology) [14, 8, 3, 15, 2]. Domain-specific languages like ΦLOG offer biologists with work-benches for the rapid exploration of ideas and experiments, without the burden of low-level coding of data and processes and interoperation between existing software tools.

In this project we investigate the *design*, *development*, and *application* of a *Domain Specific Language (DSL)*, called ΦLOG, for rapid prototyping of bioinformatic applications in the area of phylogenetic inference and evolutionary biology. Phylogenetic inference involves study of evolutionary change of traits (genomic sequences, morphology, physiology, behavior, etc.) in the context of biological entities (genes, individuals, species, higher taxa, etc.) related to each other by a phylogenetic tree or genealogy depicting the set of common ancestors. It finds important applications in areas such as study of ecology and dynamics of viruses. To be attractive to biologists, an effective DSL should provide:

(1) descriptions of the concepts and operations naturally associated with biology (e.g., data sources, types of data, transformations of the data),
(2) mechanisms allowing users to manipulate those concepts in a compact, intuitive manner at a high level of abstraction,
(3) models specialized enough to reflect the real biological processes of interest, and
(4) efficient execution mechanisms that do not require extensive intervention and programming by the end user.

Furthermore, a large class of biological models integrates information on relationships among organisms, homology of traits, and specifications of evolutionary change in traits; this commonality can be used to advantage in designing the structure of domain-specific representations and transformations of biological data. Information technology based on the major commonality evident in problems explicitly involving relationships, homology, and trait evolution can readily be expanded to incorporate a much broader range of biological models. To date no software development environment or methodology available to biologists has identified all of these elements and explicitly designed uniform solutions incorporating them. Instead, there exist a large array of mostly ad hoc

technologies. Existing tools provide monolithic interfaces (rather than libraries encapsulating the basic computational elements from which larger constructions can be built) and a black box structure, that does not provide access to the underline mechanisms and heuristics [28] used to solve biological problems.

ΦLOG is part of a comprehensive computational framework, based on agent technology, capable of harnessing local, national, and international data repositories and computational resources to make the required modeling activities feasible. Solving a typical problem in phylogenetic inference requires the use of a number of different bioinformatic tools, the execution of a number of manual steps (e.g., judging which sequence alignment for two genes is the "best"), and extra low-level coding to glue everything together (e.g., low-level scripting). An important characteristic of the ΦLOG framework is its ability to *interoperate* with the existing biological databases and bioinformatic tools commonly used in phylogenetic processing, e.g., CLUSTAL W, BLAST, PHYLIP, PAUP. These existing tools and data repositories are treated as *semantic Web services*, automatically accessed by ΦLOG to develop the solution requested by the domain expert. From a semantic point of view, these existing components are regarded as semantic algebras [23], used for defining the valuation predicates in the denotational specification of the DSL. Thus, ΦLOG provides a uniform language through which biologists can perform complex computations (involving one or more of these software systems) without much effort. In absence of such a DSL, biologists are required to perform significant manual efforts (e.g., locating and accessing tools, determine adequate input/output data formats) and to write considerable amount of glue code.

The execution model of ΦLOG is built on an agent infrastructure, capable of transparently determining the bioinformatic services needed to solve the problem and the data transformations that are required to seamlessly stream the data between such components during the execution of a ΦLOG program. Bioinformatic services are viewed as actions in situation calculus, and the problem of deriving a correct sequence of service invocations is reduced to the problem of deriving a successful plan. The agent infrastructure relies on a *service broker* for the discovery of bioinformatic services, and each agent makes use of logic-based planner for composition and monitoring of services. The framework implements typical agents' behaviors, including planning, interaction, and interoperation.

2 The ΦLOG System

ΦLOG is based on a comprehensive agent-based platform, illustrated in Figure 1.

The higher level is represented by the ΦLOG language, a DSL specifically designed for evolutionary biologists—described in Section 3.

The execution of each ΦLOG program is supported by a *DSL compiler*—described in Section 4—and an *Execution Agent*—described in Section 5. The execution agent is, in turn, composed of a *configuration component* and an *execution/monitoring component*. In this framework, bioinformatic services are viewed as *actions*, and execution of ΦLOG programs as an instance of the planning problem. The compiler translates each ΦLOG program into a *partial plan*—specifically a GOLOG program [13]; this de-

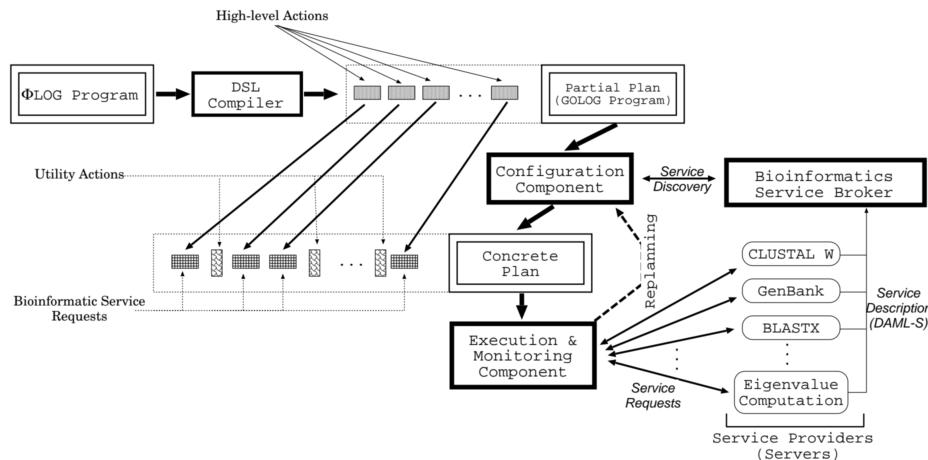

Fig. 1. System Organization

scribes the steps required to execute the *Φ*LOG program in terms of high-level actions and their sequencing. The plan is considered partial for various reasons:

(i) each high-level action has to be resolved into invocation of actual bioinformatic software tools and data sources;
(ii) interaction between successive steps in the plan may require the introduction of intermediate low-level actions (e.g., interoperation between existing tools).

The actual execution requires transformation of the partial plan into a *concrete plan*—whose (low-level) actions are actual accesses to the data repositories and execution of bioinformatic tools. This transformation is accomplished by the configuration component of the agent, via a planning process. This planning process is performed in cooperation with a *service broker*, which supplies description and location of the existing data sources and software tools. The configuration agent makes use of these services descriptions to develop the action theory needed to generate the concrete plan.

The execution of the concrete plan is carried out by the execution/monitoring component of the agent. Execution involves contacting data sources and software tools and requesting the appropriate execution steps. Monitoring is required to validate progress of the execution and re-enter the planning phase to repair eventual execution failures. In the successive sections we highlight the relevant aspects of the various components and the research challenges to be tackled.

3 The Design of the *Φ*LOG Language

In this section we propose a preliminary design of the *Φ*LOG language. More details regarding this initial design can be found in [20]. We use problems from biology as motivating examples. These examples relate to one of the most challenging problems in biology: that of determining the evolution of species. The evolution of a set of species can naturally be represented by an *evolutionary tree* with leaf nodes representing sampled species, interior nodes representing ancestors, and edges representing the ancestor-descendant relationship in the usual fashion. Given a set of related species there is a

multitude of possibilities as to how they evolved. The goal is to use the biological data to determine the most likely evolutionary history. One way to determine how a set of related species evolved is based on modeling DNA sequence data using a stochastic model of evolutionary change [28]. The evolutionary tree that "best" fits this model and data is adopted as the most likely evolutionary history. The starting point of this process is represented by the collection of similar DNA sequences for the set of species of interest from the huge set of DNA sequence data that is stored in databases like GSDB and GenBank. In its simplest form, both the set of taxa and the set of genes are completely specified, and the task is simply to determine occurrences (i.e., sequences) of the genes in the taxa. Matching sequences are determined by comparing the given genes with the sequences belonging to each taxon, and applying a set of filtering criteria. This result is typically constructed by iterating (manually or using ad-hoc scripts) the application of *similarity* search programs—e.g., the *Basic Local Alignment Search Tool (BLAST)*—using the provided genes as input and (manually) filtering the output with respect to the taxa of interest. During each iteration one of the genes is used to detect reasonable matches against a sequence database. The resulting matches have to be filtered to extract only matches relative to the taxa of interest and to remove false matches.

The successive step is to find the most likely evolutionary tree for a given set of species from the given DNA sequence data and a model of evolution [28]. The current methodology to solve this problem is to:

(i) align the input sequences—sequence alignment can be performed using a standard tool for multiple sequence alignment, such as CLUSTAL W—and
(ii) use the aligned sequences and the given model(s) to generate and rank possible phylogenetic trees for the sequences, using tools such as PHYLIP and PAUP.

In this approach, users are responsible for proper pipelined execution of all the components (including data format translation, if needed). Furthermore, most tree building software uses a very limited set of evolutionary models, and existing software considers only limited parameter optimizations for parametric models [28].

3.1 Preliminary DSL

In this section we provide a brief overview of the *Φ*LOG language. Rather than providing a comprehensive definition of the language, we will introduce the concepts and elements of the language through examples. We appeal to the users for the intuitive meaning of the keywords used in this section. For a more complete description, the interested reader is referred to [20]. In what follows, sample codes of *Φ*LOG are written in `verbatim` font.

Overall Program Structure. *Φ*LOG programs consist of modules. Each module contains a collection of global declarations and a collection of procedures. In turn, each procedure contains a sequence of declarations and a sequence of instructions. The declaration part of each procedure is used to

(i) describe the data items used by the procedure,

(ii) allow user selection of the computational components to be used during execution; and

(iii) provide parameters affecting the behavior of the different components.

Data items used in the program must be declared. Declarations are used to explicitly describe data items, by providing a name (<item name>), a description of the nature of the values that are going to be stored in it (<item type>) and eventual properties of the item. Formally, a declaration is written as

```
<item name> : <item type>
```

For example, the expression

```
gene1 : Gene ( gi | 557882 )
```

declares an entity called gene1, of type Gene, and identifies the initial value for this object—the gene which has GI number 557882 in the GenBank database.

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

```
similar: operation (BLASTX -- alignment=ungapped
                    database=drosophila matrix=BLOSUM45)
```

allows the user to explicitly configure the behavior of the language operation similar—by associating this operation with the BLAST similarity search program.

Data Types. The design of the collection of data types of ΦLOG has been driven by observing the commonalities present between various languages for biological data description proposed in the literature (e.g., BSML and NEXUS [14]). ΦLOG provides two classes of data types that can be used to create data items. The first class includes generic (non-domain specific) data types, while the second class provides a number of *Domain Specific Data Types*, which are relevant for the specific domain. E.g.,

- Sequence, Gene, and Taxon: These data types are used to describe molecular sequence data (e.g., DNA, RNA), Genes, and Taxons (e.g., species along with their traits). The sequence data type allows users to describe the sequence in different ways, e.g., by providing its description in the standard formats. Using the FASTA format one could describe a sequence as follows:

  ```
  g : Sequence (protein)
  g is FOSB_HUMAN P53539 homo sapiens (HUMAN).
       MFQAFPGDYDSGSRCSSSSPSAESQYLSSVDSFGPPT...
  ```

  Transformation between the different formats is transparent. Each object of type Gene and Taxon is characterized by a number of (optional and mandatory) attributes, including name, accession number, GI number, and sequence data [14]. The actual set of attributes depends on the detail of the description provided by the user and/or on the existing attributes present in the gene database used. For example:

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

assigns to the item `g1` the gene having GI number `gi|557882` and extracts its sequence data, which is stored in the item `se`. The `Taxon` data type is polymorphic in the sense that items of type `Taxon` can be used to represent higher-order taxa as well, which are interpreted in the language as (hierarchically organized) sets of taxa.

- `Model:` A model data type is used to describe models of evolution, used to perform inference of phylogenies [28]. In the simple case, a model is a matrix containing the individual evolutionary rates:

  $$\texttt{t : Model is (\{A} \rightarrow \texttt{(0.1)C, A} \rightarrow \texttt{(0.04)T,} \cdots\texttt{\}).}$$

  *Φ*LOG allows symbolic description of models [20] and access to standard models [28], e.g., `t1 is K80(kappa).`

*Φ*LOG also provides a number of polymorphic data types which are used to aggregate in different ways collection of entities. *Φ*LOG allows operations on `Trees` (described using Newick format) and `Sets`. In particular, various externally accessible sources of information are mapped in the DSL as sets of elements. For example, external databases, e.g., GenBank, can be accessed in the language using traditional set operations, e.g.,

```
{name(x) | x: Gene, x in GenBank}
```

In *Φ*LOG a `Map` represents a function which maps elements of a domain into elements of another domain. The mapping can be specified either as an enumeration of pairs or using an intensional definition—i.e., the mapping is described via properties. For example,

```
match ( tax : Taxon, x : Gene ) : Map
    ( S is { y : Sequence | y in genes(tax),
                score(similar(x,y)) > threshold }
     if S is empty
       then match is undefined
       else match is (any y in S) )
```

defines a `Map`, named `match`, which is a function. In this example, we have a function which maps a taxon and a gene to a sequence. More precisely, given a taxon `tax` and a gene `x`, `match` returns a sequence of a gene which is sufficiently similar to `x`. In the above code, the first line declares that `match` is a function (i.e., of the type `Map`) with two input parameters; the rest specifies how the function is computed. First, a set `S` of sequences which are sufficiently similar to `x` is computed. If this set is empty then `match` is undefined (the ‘then’ clause); otherwise, it is defined as an arbitrary sequence in `S` (the ‘else’ clause). In computing `S`, the `similar` operation is expected to return a data item measuring the similarity of the two sequences; in this example we expect an attribute called `score` to be present.

Maps can be used in various ways—in the previous example, if `Ta` is a set of taxa, then `match(Ta,seq)` is a set of sequences (constructed according to the map definition), while an expression such as `select X (seq1 is match(X,seq2))` searches for a taxon `X` containing the sequence `seq1` which is similar to sequence `seq2`. Other polymorphic types include the ability to describe probability distributions and homologies [20].

Control Structures. The language provides two levels of control constructs. At the higher level, control of the execution is expressed using declarative constructs, such as function applications and quantifications. Existential quantification (`select`) is used to express search tasks, while universal quantification (`forall`) can be used to express iterations and global properties. In this respect, the structure of the language closely resembles the structure of many functional languages. For example, the code

```
select a in Trees
    forall b in Trees
     likelihood(a,model) ≥ likelihood(b,model)
```

selects a tree `a` from the collection of trees `Trees` that has the maximum likelihood.

At the lower level, the language provides control constructs which are closer to those in traditional programming languages, such as `if-then-else` conditional statements and `repeat` iterative constructs. These are mostly used by users who want to customize the basic operations of the language (as described in the following example). The previous example can be rewritten as (`best, t` are of type `Tree`)

```
best in Trees
repeat (t in Trees)
         if    (likelihood(t,model)>likelihood(best,model))
               then best is t
```

The first statement selects as initial value for `best` an arbitrary element of `Trees`, and the loop performs an iteration for each element in `Trees`. The language provides also the capability of asserting constraints on the computation. These constraints are assertions that have to be satisfied at any point during the computation. Constraints are asserted using the `constraint` statement. E.g., if we want the computation to generate a tree `t` whose root has only two children, with branch lengths 10 and 20, then we can use the statement

```
constraint : t is (X : label1 , Y : label2)Z and
             label1 is 10 and label2 is 20
```

The execution model adopted by the language provides the user with both batch and interactive execution. Under batch mode, programs are compiled and completely executed. Under interactive mode, programs are executed incrementally under user supervision. The user is allowed to select breakpoints in the execution, run the program statement by statement, and modify *both the data and the program* on the fly. For example, the user is allowed to introduce new constraints *during* the execution, thus modifying on the fly the behavior of the computation. Another feature that *Φ*LOG provides is the ability to make data items *persistent*, allowing users to create with no extra effort databases containing the results of the computations, and to share partial results.

3.2 Some Examples Coded in *Φ*LOG

Let us start by looking at how we can express the problem of, given a collection of taxa and a collection of genes, determining in which taxa those genes occur. Both the set of taxa and the set of genes can be either explicitly provided by the user or the result of

some computation. Regarding the selection of taxa, we need to determine a set of taxa of interest, eventually involving higher-order taxa. This set could be the result of some computation, e.g., to select 40 taxa out of a given higher-order taxon:

```
Tax is {y : Taxon | y in murinae} and |Tax|=40
```

or, given a collection of higher-order taxa (InputSet), select a taxon from each of them:

```
Tax is union ( x in InputSet , t is (any in x) ).
```

Regarding the selection of the genes of interest, this can be either a user defined collection of genes or it could be itself the result of some computation. E.g., to select all genes from a given set of taxa (T) which contain a certain name:

```
Gen is {x : Gene | taxa in T, x in genes(taxa),
                   name(x) contains "Adh" }
```

Once a set of taxa (Tax) and a set of genes (Gen) have been identified, then we would like to determine occurrences of the identified genes in the taxa of interest—this can be accomplished using the match map defined earlier. The selection of the components of the map requires a filtering of the result of the similarity search. E.g., if we are interested in defining the map in order to produce the sequence with the longest aligned region, then we simply replace the any statement with

```
w in S and forall z in S
  length_alignment(similar(y,w)) ≥
             length_alignment(similar(y,z))
```

As part of the definition of the Sequence data type, the language provides an operation, align, which provides the sequence alignment capabilities. The align operation accepts a single argument which represents the set of sequences to be aligned. The result of the operation is a set of sequences, containing the original sequences properly expanded to represent the desired alignment. The behavior of align can be customized by the user similarly to what is described in the case of similar. E.g., the declaration:

```
align : operation (CLUSTAL W -- model=PAM)
```

asserts that the align operation should be performed by accessing the CLUSTAL W software with the appropriate parameters. We envision generalizing the behavior of align to produce as result not just a set of aligned sequences but a more general object—a data item of type Homology. The next step requires the definition of the model which is going to be used to describe evolutionary rates—i.e., a data item of the type Model. At the highest level, we can assume the presence of a build_tree operation which directly interfaces to dedicated tools for inference of phylogenies:

```
build_tree : Operation ( DNAML -- )
t          : Tree is build_tree(Seqs,model1)
```

This reflects the current standard approach, based on the development of a tree using a single model across the entire sequence and the entire tree [28]. The operation build_tree can be redesigned by the user whenever a different behavior is required. A simple declarative way of achieving this is as follows: given a list list

```
constraint : forall [X,Y] in list
             (likelihood(X,model) ≥likelihood(Y,model))
list is      [ t : Tree | t in Tree(Seqs) ]
```

which expresses the fact that `list` contains all the trees over the sequences in the set `Seqs`, sorted according to the likelihood of each tree under the model `model`. The best tree is the first in the list. A finer degree of control can be obtained by switching to the imperative constructs of the language and writing explicit code for the search. E.g., the following ΦLOG code picks the best of the first 1000 trees generated:

```
t    is     initialTree(Seqs)
best is     t
     repeat (i from 1 to 1000)
            t is nextTree(t,Seqs,model)
            if likelihood(t,model)>likelihood(best,model)
                  then best is t
```

By providing different definitions of the operations `initialTree` and `nextTree` it is possible to customize the search for the desired tree. The same mechanisms can be used to select a set of trees instead of just one.

Given a set of trees `Trees` computed from a set of sequences and given a model `model`, we can construct a relative likelihood for the given set of trees:

```
prob :   Probability(Trees)
total is summation(x in Trees,likelihood(x,model))
         forall x in dom(prob)
               prob(x) is likelihood(x,model) / total
```

This model can be easily extended to accommodate a distribution of models instead of an individual model. This allows us to use expected likelihood for the evaluation and selection of the trees. Assuming a finite collection of models `Models` and an associated probability distribution `prob`, we can replace the `likelihood` function by the `fit` function defined below:

```
fit (t : Tree) : map
    (fit is sum(m in Models,prob(m)*likelihood(t,m)))
```

4 Compilation of ΦLOG Programs

The goal of the ΦLOG compiler is to translate ΦLOG programs—manually developed by a biologist or developed via high-level graphical interfaces [20]—into partial plans. The components of the partial plan are high-level actions, extracted from an ontology of bioinformatic operations; each high-level operation will be successively concretized into one or more invocations of bioinformatic services (by the execution agent).

4.1 Bioinformatic Services

Data sources and software tools employed to accomplish phylogenetic inference tasks are uniformly viewed by ΦLOG as *bioinformatic services*. Each service provides a uniform interface to the data repository or software tool along with a description of the functionalities of the service (e.g., inputs, outputs, capabilities). Service descriptions are represented using a standard notation for service description in the semantic web (specifically DAML-S [5]). Service providers register their services with the *services broker*. The task of the broker is to maintain a directory of active services (including the location of the service and its description) and to provide matchmaking services between service descriptions and service requests.

The creation and management of service descriptions require the presence of a very refined *ontology*, describing all entities involved in service executions. Ontologies provide an objective specification of domain information, representing a community-wide consensus on entities and relations characterizing knowledge within a domain. The broker employs the ontology to instantiate high-level requests incoming from configuration agents into actual service requests. Considerable work has been done in the development of formal ontologies for biological concepts and entities [26, 1]. Our project will build on these efforts, taking advantage of the integrated description of biological entities provided in these existing proposals. Nevertheless, this project aims at covering aspects that most of these ontologies currently do not provide:

(i) Description of an actual hierarchy of *bioinformatic operations*, their relationships, and their links to biological entities; each operation is described in terms of input and output types as well as its effect.

(ii) Description of an actual hierarchy for *bioinformatic types* and *bioinformatic data representation formats*, their relationships, and the links to biological entities and to the bioinformatic applications.

Thus, our objective is to concretize biological ontologies by introducing an ontology level describing the data formats and transformations that are commonly employed in phylogenetic inference, and linking them to the existing biology ontologies. This additional level (i.e., an ontology for bioinformatic tools) is fundamental to effectively accomplish the instantiation of a partial plan into a concrete plan—e.g., automatically selecting appropriate data format conversion services for interoperation.

Standard semantic web services description languages—i.e., DAML-S—have been employed for the development of these ontologies.

Service descriptions as well as the bioinformatic ontologies are maintained and managed by a service broker. In this project, the service broker is developed using the *Open Agent Architecture (OAA)* [16]. This will allow us to apply previously developed techniques in web services composition and GOLOG programs's execution and monitoring in [17] in our application. The advantages of this approach have been discussed in [18]. Currently, we have developed a minimal set of service descriptions that will allow us to develop simple ΦLOG programs that will be used as testbeds for the development of other components of the systems. We use OAA in this initial phase as it provides us the basic features that we need for the development of other components. In the later phases

of the project, we will evaluate alternative architectures with similar capabilities (e.g. InfoSleuth [7]) or multi-agent architectures (e.g. RETSINA [27] or MINERVA [12]).

4.2 ΦLOG Compiler

The derivation of the ΦLOG compiler has been obtained using a semantic-based framework called *Horn-Logic Denotations (HLD)* [9]. In this framework, the syntax and semantics of the DSL is expressed using a form of denotational semantics and encoded using an expressive and tractable subset of first-order logic (Horn clauses). Following traditional denotational semantic specifications, in HLD a DSL $\mathcal{L}$ is described by three components: *(i)* syntax specification—realized by encoding a context free grammars as Horn clauses (using the Declarative Clause Grammars commonly used in logic programming); *(ii)* interpretation domains—described as logic theories; *(iii)* valuation functions—mappings from syntax structures to interpretation domains (encoded as first-order relations). In our specific application, the interpretation domains are composed of formulae in an action theory [21], describing properties and relationships between bioinformatic services. The action theories are encoded using situation calculus [21]—see also Section 5—and they are extracted by the compiler from the services ontologies described in the previous section. The semantic specification also allows for rapid and correct implementations of the DSL. This is possible thanks to the use of an encoding in formal logic and the employment of logic-based inference systems—i.e., the specifications are *executable*, automatically yielding an interpreter for the DSL. Moreover, the Second Futamura Projection [10] proves that compiled code can be obtained by *partially evaluating* an interpreter w.r.t. a source program. Thus, our DSL interpreter can be partially evaluated (using a partial evaluator for logic programming [22]) w.r.t. a program expressed in the DSL to obtain compiled code [9]. In our context, the final outcome of the process is the automatic transformation of specifications written in ΦLOG to programs written in GOLOG.

The semantic specification can also be extended to support verification and debugging: the denotational specifications provide explicit representation of the state [23, 9], and manipulation of such information can be expressed as an alternative semantics of the DSL—i.e., as an *abstract semantics* of the DSL [9]. This verification process is also possible thanks to the existence of *sound and complete* inference systems for meaningful fragments of Horn clause logic [19]. Given that the interpreter, compiler, and verifiers are obtained directly from the DSL specification, the process of developing and maintaining the development infrastructure for the DSL is very rapid. Furthermore, syntax and semantic specifications are expressed in a *uniform* notation, backed by effective inference models.

5 Execution Agent

5.1 ΦLOG's Program Execution as Planning and Plan Execution Monitoring

A ΦLOG program specifies the general steps that need to be performed in a phylogenetic inference application. Some steps can be achieved using built-in operations—this

is the case for the primitive operations associated to the various data types provided by ΦLOG—e.g., compose two sets using a union operation or selecting the members of a set that satisfy a certain condition. Other steps might involve the use of a bioinformatic service, e.g., CLUSTAL W for sequence alignment, BLAST for similarity search, PAUP for phylogeny construction, etc. In several cases, intermediate steps, not specified by the compiler, are required; this may occur whenever the ΦLOG program does not explicitly lay out all the high-level steps required, or whenever additional steps are required to accomplish interoperability between the bioinformatic services. For example, the sequence alignment formats provided by CLUSTAL W cannot be used directly as inputs to the phylogenetic tree inference tool PAUP (which expects a Nexus file as input). As such, a ΦLOG program could be viewed as a skeleton of an application rather than its detailed step-by-step execution. Under this view, *execution of ΦLOG programs could be viewed as an instance of the planning and plan execution monitoring problem*. This will allow us to apply the techniques which have been developed in that area to implement the ΦLOG engine. We employ the approach introduced in [17] in developing the ΦLOG system. In this approach, each bioinformatic service is viewed as an *action* and a phylogenetic inference application as a GOLOG program [13].

We will now review the basics of GOLOG and the agent architecture that will be used in the development of the ΦLOG system. Since GOLOG is built on top of the situation calculus (e.g.,[21]), we begin with a short review of situation calculus.

Situation Calculus. The basic components of the situation calculus language, following the notation of [21], include a special constant S_0, denoting the initial situation, a binary function symbol do where $do(a, s)$ denotes the successor situation to s resulting from executing the action a, fluent relations of the form $f(s)$, denoting the fact that the fluent f is true in the situation s, and a special predicate $Poss(a, s)$ denoting the fact that the action a is executable in the situation s.

A dynamic domain can be represented by a theory containing: (i) axioms describing the initial situation S_0; (ii) action precondition axioms (one for each action a, characterizing $Poss(a, s)$); (iii) successor state axioms (one for each fluent F, stating under what condition $F(\boldsymbol{x}, do(a, s))$ holds, as a function of what holds in s); (iv) unique name axioms for the primitive actions; and some foundational, domain independent axioms.

GOLOG. The constructs of GOLOG [13] are:

α	*primitive action*
$\phi?$	*wait for a condition*
$(\sigma_1; \sigma_2)$	*sequence*
$(\sigma_1 \vert \sigma_2)$	*choice between actions*
$\pi x.\sigma$	*choice of arguments*
σ^*	*nondeterministic iteration*
if ϕ **then** σ_1 **else** σ_2	*synchronized conditional*
while ϕ **do** σ	*synchronized loop*
proc $\beta(\boldsymbol{x})\sigma$	*procedure definition*

The semantics of GOLOG is described by a formula $Do(\delta, s, s')$, where δ is a program, and s and s' are situations. Intuitively, $Do(\delta, s, s')$ holds whenever the situation s' is a terminating situation of an execution of δ starting from the situation s. For example, if $\delta = a; b \mid c; f?$, and $f(do(b, do(a, s)))$ holds, then the GOLOG interpreter will determine that $a; b$ is a successful execution of δ in the situation s.

The language GOLOG has been extended with various concurrency constructs, leading to the language *ConGolog* [6]. The precise semantic definitions of GOLOG and ConGolog can be found in [13, 6]. Various extensions and implementations of GOLOG and ConGolog can be found at `http://www.cs.toronto.edu/~cogrobo` web site. An *answer set programming* interpreter for GOLOG has been implemented [24].

Bioinformatic Services as Actions and ΦLOG's Programs as GOLOG Programs. A bioinformatic service is a web service—i.e., a computational service accessible via the Web. Adopting the view of considering web services as actions for Web service composition application [17], we view each bioinformatic service as an action in situation calculus. Roughly speaking, an action description of a web service consists of the service name, its invocation's description, and its input and output parameters with their respective formats. Let $\mathcal{D}$ be the set of actions representing the bioinformatic services. Under this framework, the problem of combining bioinformatic services to develop a phylogenetic inference application is a planning problem in $\mathcal{D}$. Since $\mathcal{D}$ contains all the external operations that a user of ΦLOG can use, each ΦLOG program P can be compiled into a GOLOG program in the $\mathcal{D}$ language by:

- Introducing variables and fluents representing the variables of the ΦLOG program.
- Replacing each instruction `V is operation(`$\boldsymbol{x}$`)` in the ΦLOG program with the action $\tau(\texttt{operation})(\boldsymbol{x}, V)$, where τ is a mapping that associates ΦLOG operations to invocations of bioinformatic services. The ΦLOG compiler implements the τ operation through the following steps:
  - terms in the ΦLOG program are used to identify the set of *high-level actions* requested by the ΦLOG programmer. The process is accomplished by identifying relevant entries in the hierarchy of bioinformatic operations (see Section 4.1).
  - high-level operations are used to query the services broker and retrieve the description of relevant registered services that implement the required operations.
- Replacing the control constructs such as **if-then, for-all**, etc. in the ΦLOG program with their corresponding constructs in GOLOG.

In turn, the description of the various services retrieved from the services broker (DAML-S descriptions) are converted [17] into action precondition axioms and successor state axioms. This translation will be achieved by the compiler of the ΦLOG system. The GOLOG program obtained through this translation may contain the *order* construct (denoted by $\sigma_1 : \sigma_2$), an extended feature of GOLOG suggested in [17], and used to describe a partial order between parts of the program. This feature is essential in our context; the ΦLOG program determines the ordering between the main steps of the computation (expressed using the order construct), while the configuration agent may need to insert additional intermediate steps to ensure interoperation between services and exe-

cutability of the plan. E.g., if a ΦLOG program indicates the need to perform a sequence alignment (mapped to the `clustalw` service) followed by a tree construction (mapped to the `paup` service), then the GOLOG program will contain $\texttt{clustalw : paup}$; the agent may transform this into the plan $\texttt{clustalw; parse_nexus; paup}$, by inserting a data format convertion action (`parse_nexus`). Using a GOLOG interpreter we can find different sequences of services which can be executed to achieve the goal of the ΦLOG program, provided that $\mathcal{D}$ is given. Precise details in the translation and execution monitoring under development.

5.2 Configuration Component

The configuration component is in charge of transforming a partial plan (expressed as a GOLOG program with extended features, as described in [17]) into a concrete plan (*instantiation*). This process is a *planning problem* [24], where the goal is to develop a concrete plan which meets the following requirements:

(i) each high-level action in the partial plan is instantiated into one or more low-level actions in the concrete plan, whose global effect correspond to the effect of the high-level action (i.e., the τ operation mentioned earlier);
(ii) successive steps in the concrete plan correctly interoperate.

This process is intuitively illustrated in Example 1. The configuration component makes use of the high-level actions in the partial plan to query the services broker and obtain lists of concrete bioinformatic services that can satisfy the requested actions. The querying is realized through the use of the bioinformatic tools ontology mentioned earlier.

The configuration component attempts to combine these services into an effective concrete plan—i.e., an executable GOLOG program. The process requires fairly complex planning methodologies, since:

- the agent may need to repeatedly backtrack and choose alternative services and/or add intermediate additional services (e.g., filtering and data format transformation services) to create a coherent concrete plan;
- planning may fail if some of the requested actions do not correspond to any service or services cannot be properly assembled (e.g., lack of proper interoperation between data formats); sensing actions may be employed to repair the failure, e.g., by cooperating with the user in locating the missing service;
- planning requires the management of *resources*—e.g., management of a budget when using bioinformatic services with access charges;
- planning requires *user preferences*—e.g., choice of preferred final data formats.

Most of these features are either readily available in GOLOG or they have been added by the investigators [17, 24, 25] to fit the needs of similar planning domains.

Example 1. Consider the original simple ΦLog program (g is of type $Genes$, s of type $Alignment(dna)$ and t of type $Tree(dna)$):

```
g is { x : Gene| name(x)  contains  ``martensii''}
s is align(g)
t is phylogenetic_tree(s)
```

The high-level plan detected by the compiler:

```
database_search(gene,[(GeneName,''martensii'')], o1) ;
                         sequence_alignment(dna, o1, o2) ;
                          phylogenetic_tree(dna, o2, o3) ;
                                         display_output(o3)
```

The action theory derived from the agent broker will include:

$genebank(''GeneName = martensii'', o1)$ **causes** $genes(o1, G)$ **if** $\ldots$
$clustalw(o1, o2)$ **causes** $alignment(dna, o2, A)$ **if** $sequences(dna, o1, S), \ldots$
$dnaml(o2, o3)$ **causes** $tree(dna, o3, T)$ **if** $alignment(dna, o2, A), \ldots$

Additional actions are automatically derived from the type system of the language; for example, `Gene` is seen as a subclass of `DNA_Sequence`, which provides an operation of the type:

$$gene_to_dnasequence(o) \textbf{ causes } sequence(dna, o) \textbf{ if } gene(o)$$

The operation will be associated to built-in type conversion actions. Finally, the high level plan will have to be replaced by a GOLOG program, where each high-level action should be replaced by a choice of actions construct (a choice from the set of the services implementing such high-level action), e.g.,

```
genebank(''Gene Name = martensii'',o1) | ··· :
                       clustalw(o1, o2) | ··· :
                           dnaml(o2,o3) | ··· :
                       display_output(o3) ;
                           ? tree(dna,o3).
```

5.3 Execution and Monitoring Agent

Once a concrete plan has been developed, the execution/monitoring component of the agent proceeds with its execution. Executing the concrete plan corresponds to the creation and execution of the proper service invocations corresponding to each low-level action. Each service request involves contacting the appropriate service provider—which can be either local or remote—and supply the provider with the appropriate parameters to execute the desired service. The monitoring element supervises the successful completion of each service request. In case of failure (e.g., a timeout or a loss of connection to the remote provider), the monitor takes appropriate repair actions. Repair may involve either repeating the execution of the service or re-entering the configuration component. 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 of the part of the concrete plan executed before the failure.

6 Discussion and Conclusions

6.1 Technology

The development of this framework employs a combination of novel and existing software technology. DAML-S is used as representation format for the description of services. In the preliminary prototype, the ontologies for bioinformatic services and for bioinformatic data formats have been encoded using logic-based descriptions—specifically, using the OO extensions provided by SICStus Prolog.

The reasoning part of the agent (configuration, execution, and monitoring) is based on *situation calculus*. This is in agreement with the view adopted by the semantic web community—Web Services, encoded in DAML-S, can be viewed as actions. GOLOG is employed as language for expressing the partial plans derived from ΦLOG programs, as well as describing the complete plans to be executed. In this work we make use of a GOLOG interpreter encoded in answer set programming [11]. The advantage of this approach is that it allows us to easily extend GOLOG to encompass the advanced reasoning features required by the problem at hand—i.e., user preferences [25] and planning with domain specific knowledge [24].

6.2 Related Work

An extensive literature exists in the field of DSL [29]. Design, implementation, and maintenance of DSLs have been identified as key issues in DSL-based software development and various methodologies have been proposed (e.g., [4, 29]. HLD is the first approach based on logic programming and denotational semantics, and capable of completely specifying a DSL in a uniform executable language.

Various languages have been proposed to deal with the issue of describing biological data for bioinformatic applications. Existing languages tend to be application-specific and limited in scope, they offer limited modeling options, are mostly static, and are poorly interconnected. Various efforts are undergoing to unify different languages, through markup languages (e.g., GEML and BSML) and/or ontologies [26]. Some proposals have recently emerged to address the issues of interoperability, e.g., [3, 2].

The work that comes closest to ΦLOG includes programming environments which allows scientists to *write programs* to perform computational biology tasks. Examples of these include TAMBIS [3], Darwin [8] and Mesquite [15]. They combine a standard language (imperative in Darwin, visual in Mesquite) with a collection of modules to perform computational biological tasks (e.g., sequence alignments). Both Darwin and Mesquite provide only a small number of models that a biologist can use to create bioinformatic applications, and they both rely on a "closed-box" approach. The modules which perform the basic operations have been explicitly developed as part of the language and there is little scope for integration of popular bioinformatic tools. TAMBIS provides a knowledge base for mapping graphically expressed queries to accesses of a set of bioinformatic data sources.

6.3 Conclusions and Future Work

In this paper we presented a brief overview of the ΦLOG project, aimed at the development of a domain specific framework for the rapid prototyping of applications in evolutionary biology. The framework is based on a DSL, that allows evolutionary biologists to express complex phylogenetic analysis processes at a very high level of abstraction. The execution model of ΦLOG relies on an agent infrastructure, capable of automatically composing and monitoring the execution of bioinformatic services, to accomplish the goals expressed in the original ΦLOG program. The framework is currently under development as a collaboration between researchers in Computer Science, Biology, and Biochemistry at NMSU.

Acknowledgments. The authors wish to thank the anonymous referees for their helpful comments. Research has been supported by NSF grants EIA-0130887, CCR-9875279, HRD-9906130, EIA-0220590, and EIA-9810732.

References

1. Bio-Ontologies Consortium. `www.bioontology.org`.
2. Standardizing Biological Data Interchange Through Web Services. `omnigene.sourceforge.net`, 2001.
3. P. Baker et al. Transparent Access to Bioinformatics Information Sources. *Intelligent Systems for Molecular Biology*, 1998.
4. J. Bentley. Programming pearls: Little languages. *Communications ACM*, 29(8), 1986.
5. DAML-S Coalition. DAML-S: Semantic markup for Web services. *Int. Semantic Web Working Symposium*, 2001.
6. 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.
7. J. Fowler, B. Perry, M. Nodine, and B. Bargmeyer. Agent-Based Semantic Interoperability in InfoSleuth. SIGMOD Record 28:1, March, 1999, pp. 60-67.
8. G. Gonnet and M. Hallet. *Darwin 2.0.* ETH-Zurich, 2000.
9. G. Gupta and E. Pontelli. A Horn denotational framework for specification, implementation, and verification of DSLs. In *Logic Programming and Beyond*, Springer Verlag, 2002.
10. N. Jones. Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.
11. V. Lifschitz. Answer set planning. *ICLP*, 23-37, MIT Press, 1999.
12. J. A. Leite, J. J. Alferes and L. M. Pereira. MINERVA - A Dynamic Logic Programming Agent Architecture. In *Intelligent Agents VIII*, pages 141-157, Springer-Verlag, 2002.
13. H. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. Scherl. GOLOG: A logic programming language for dynamic domains. *Journal of Logic Programming*, 31(1-3):59–84, 1997.
14. D. R. Maddison et al. NEXUS: An Extensible File Format for Systematic Information. *Syst. Biol.*, 464(4), 1997.
15. W. Maddison. Mesquite: A Modular System for Evolutionary Analysis. U. Arizona, 2000.
16. D. L. Martin, A. J. Cheyer, and D. B. Moran. The Open Agent Architecture: a Framework for Building Distributed Software Systems. *Applied Artificial Intelligence*, 13:91–128, 1999.
17. S. McIlraith and T.C. Son. Adapting Golog for Composition of Semantic Web Services. In *International Conference on Principles of Knowledge Representation and Reasoning*, 482–493, Morgan Kaufmann Publisher, 2002.

18. S. McIlraith, T.C. Son, and H. Zeng. Semantic Web Services. IEEE Intelligent Systems. Special Issue on the Semantic Web. March/April, 16(2):46-53, 2001.
19. I. Niemelä and P. Simons. An Implementation of the Stable Model Semantics. In *LPNMR*, 420–429, Springer Verlag, 1997.
20. E. Pontelli et al. Design and Implementation of a Domain Specific Language for Phylogenetic Inference. In *Journal of Bioinformatics and Computational Biology*, 1(2):1–29, 2003.
21. R. Reiter. *Knowledge in Action.* MIT Press, 2001.
22. D. Sahlin. *An Automatic Partial Evaluator for Prolog*. PhD, Uppsala, 1994.
23. D. Schmidt. *Denotational Semantics.* W.C. Brown, 1986.
24. T.C. Son, C. Baral, and S. McIlraith. Planning with Different Forms of Domain-Dependent Control Knowledge. Approach. *Int. Conf. on Logic Progr. and Nonmonotonic Reasoning*, 226–239, Springer Verlag, 2001.
25. T.C. Son and E. Pontelli. Reasoning about actions in prioritized default theory. In *JELIA*, 369–381, Springer Verlag, 2002.
26. R. Stevens. Bio-Ontology Reference Collection. `cs.man.ac.uk/~stevens/onto-publications.html`.
27. K. Sycara, M. Paolucci, M. van Velsen, and J. Giampapa. The RETSINA MAS Infrastructure. In *Autonomous Agents and MAS*, 7(1-2), 2003 (to appear).
28. D. L. Swofford et al. Phylogenetic inference. *Molecular Systematics*, Sunderland, 1996.
29. A. van Deursen et al. DSLs: an Annotated Bibliography. `www.cwi.nl/~arie`, 2000.