EDITtoTrEMBL: A Distributed Approach to High-Quality Automated Protein Sequence Annotation

Steffen Möller*, Ulf Leser†, Wolfgang Fleischmann and Rolf Apweiler
European Bioinformatics Institute, Hinxton, UK
{*moeller,leser,wfl,apweiler*}*@ebi.ac.uk*

June 14, 1998

Abstract

Many databases in molecular biology face the problem that the ever increasing rate of data production can not be handled any more with traditional methods, especially human curation. Therefore a number of projects are currently investigating methods for automated sequence annotation. This paper describes the EBI's approach to tackle this problem for protein sequences by integration of arbitrary analysis programs into a distributed and highly flexible environment. Our software framework allows an individual treatment of sequences depending on their particular properties, which is achieved through a high-level description of the pre-conditions and capabilities of analyzing modules. This does not only improve the overall performance of the annotation process, as unnecessary steps are avoided, but also enhances its quality since dependencies between different modules are taken into account. We have implemented a prototype and will soon start to use it in the production of new TrEMBL releases.

*Corresponding author: EMBL Outstation, EBI, Hinxton, Cambridge CB10 1SD, United Kingdom. Tel: +44 1223 494670, Fax: +44 1223 494468

†Permanent address: Technical University Berlin, Fachbereich 13 - CIS, Einsteinufer 17, D-10587 Berlin

Keywords

Database, Automated Sequence Analysis, Automation, Annotation, SWISS-PROT, TrEMBL, Distribution, Protein Sequence

1 Introduction

Generally there are two different approaches of molecular biology databases to the problem of sequence annotation. One possibility is to archive the annotation of the submitting party. The other possibility is to annotate the sequences with a team of professional human annotators. SWISS-PROT, a high-quality database for protein sequence data, has traditionally followed the second route ([6]). However, the ever increasing amount of data creates the need for new techniques to complement manual curation. To address this problem, the database TrEMBL (Translation of EMBL) was created as a supplement to SWISS-PROT to store all coding sequences (CDS) in the EMBL nucleotide sequence database ([15]) that are not already integrated in SWISS-PROT. The concept of SWISS-PROT + TrEMBL allows the provision of a comprehensive protein sequence database without lowering the editorial standards of SWISS-PROT, reflected in a high level of annotation, tight integration with other databases and a minimal level of redundancy.

The sequence alone is of limited value for most scientific purposes. We therefore enrich each entry in TrEMBL through automatic annotation. This means that every TrEMBL entry is analysed by a set of programs. The output of these programs is then used to derive new and improved annotation. Apart from providing valuable information for researchers, automated annotation also greatly simplifies the work of the human editor who will eventually finalize the annotation manually to enable the integration of the entry into SWISS-PROT.

A number of projects world-wide focus on automatic sequence annotation. For an overview, see for instance several contributions in [9] and [14]. The aim is to provide a stable framework where different analyzing programs can be integrated in a plug-and-play manner. Since both the number of such programs and the amount of data is increasing rapidly, certain issues for such frameworks are getting more and more important:

- It is vital that all programs are carefully checked by experts in the field before they are integrated. This is the only way to guarantee biologically meaningful annotation.

- The integration of arbitrary analysis programs must be possible at ease, allowing dynamic reconfiguration and recovery in case of a failing module. Additionally, it should be possible to integrate remote services which could for instance be offered by third-parties via the Internet.

- It is necessary to include mechanisms for the automatic distribution of processes, suitable for dynamic load balancing in a farm of workstations. The inclusion of different hardware platforms and operating systems allows the integration of programs that are only available on specific platforms.

- The system should treat sequences individually to avoid false or unnecessary processes. Most analysis programs require certain properties to be present in a sequence. If these are absent, starting the program is not necessary and can even produce false results.

- It is necessary to take interdependencies between analysis programs into account. The order in which they are applied is important, as the output of one program might be necessary as input for another. In the presence of cyclic dependencies programs might have to be started more than once.

- As programs use the output of other programs as their input, it is necessary that data exchanged is transformed appropriately to ensure semantic consistency. This implies the translation of keywords using dictionaries and controlled vocabularies.

- The results of different analysing programs are often overlapping as they compute the same type of information. In these cases the redundancy in the output should be removed. Overlapping tasks can also lead to inconsistent results. Even though their automatic resolution is usually impossible, such conflicts need to be clearly flagged.

In this paper we describe a novel approach to the problem of automated sequence annotation, EDITtoTrEMBL, the *Environment for Distributed Information Transfer to TrEMBL*, addressing all of these issues. EDITtoTrEMBL treats the automation of annotation as a workflow problem ([11]). It basically provides a flexible software framework for arbitrary analysis programs, developed in-house or by the international research community. To achieve an ap-

```
ID   Q12618          PRELIMINARY;   PRT;   476 AA.
AC   Q12618;
DT   01-NOV-1996 (TREMBLREL. 01, CREATED)
DT   01-NOV-1996 (TREMBLREL. 01, LAST SEQUENCE UPDATE)
DT   01-JUN-1998 (TREMBLREL. 06, LAST ANNOTATION UPDATE)
DE   ACYL-COA DESATURASE 1 (EC 1.14.99.5) (STEAROYL-COA DESATURASE 1)
DE   (FATTY ACID DESATURASE 1).
GN   OLE1.
OS   AJELLOMYCES CAPSULATA (HISTOPLASMA CAPSULATUM).
OC   EUKARYOTA; FUNGI; ASCOMYCOTINA; PLECTOMYCETES; GYMNOASCALES.
RN   [1]
RP   SEQUENCE FROM N.A.
RC   STRAIN=DOWNS;
RX   MEDLINE; 96122844.
RA   GARGANO S., DI LALLO G., KOBAYASHI G.S., MARESCA B.;
RL   LIPIDS 30:899-906(1995).
CC   -!- CATALYTIC ACTIVITY: STEAROYL-COA + AH(2) + O(2) = OLEOYL-COA + A +
CC       2 H(2)O.
CC   -!- COFACTOR: IRON.
CC   -!- SUBCELLULAR LOCATION: ENDOPLASMIC RETICULUM.
DR   EMBL; X85963; G757860; -.
DR   PROSITE; PS00476; FATTY_ACID_DESATUR_1; 1.
KW   OXIDOREDUCTASE; FATTY ACID BIOSYNTHESIS; IRON; MEMBRANE;
KW   ENDOPLASMIC RETICULUM.
SQ   SEQUENCE   476 AA;  53790 MW;  E175E586 CRC32;
     MALNEAPTAS PVAETAAGGK DVVTDAARRP NSEPKKVHIT DTPITLANWH KHISWLNVTL
     IIAIPIYGLV QAYWVPLHLK TALWAVVYYF MTGLGITAGY HRLWAHCSYS ATLPLKIYLA
     AVGGGAVEGS IRWWARGHRA HHRYTDTDKD PYSVRKGLLY SHIGWMVMKQ NPKRIGRTEI
     TDLNEDPVVV WQHRNYLKVV IFMGIVFPML VSGLGWGDWF GGFIYAGILR IFFVQQATFC
     VNSLAHWLGD QPFDDRNSPR DHIVTALVTL GEGYHNFHHE FPSDYRNAIE WHQYDPTKWT
     IWIWKQLGLA YDLKQFRANE IEKGRVQQLQ KKIDQRRAKL DWGIPLEQLP VIEWDDYVDQ
     AKNGRGLIAI AGVVHDVTDF IKDHPGGKAM INSGIGKDAT AMFNGGVYNH SNAAHNQLST
     MRVGVIRGGC EVEIWKRAQK ENKEVESVRD EYGNRIVRAG AQVTKIPEPI TTADAA
//
```

Figure 1: An example TrEMBL entry

propriate treatment of individual sequences, execution of these programs is controlled by high-level descriptions of the conditions that must be fulfilled to make their application meaningful. Using these descriptions, a sequence of analysis steps is deduced dynamically at runtime.

2 SWISS-PROT+TrEMBL

2.1 Format

TrEMBL and SWISS-PROT are currently distributed as flat-file databases. They consist of a large number of structurally homogeneous *entries*, each representing one protein sequence together with its annotation. The annotation describes the function of the protein, post-translational modifications (phosphorylation, acetylation, ...), domains and sites, secondary and quaternary structure, similarities to other proteins, diseases associated with deficiencies, sequence conflicts, variants, etc.

Each entry consists of a number of lines, each starting with a two letter identifier, the *line tag* (see fig. 1). The line tag identifies the content or type of the line. Important line types are comment lines (tag `CC`), the feature table (`FT`) and keyword lines (`KW`). Each entry has a name, which is stored in the `ID` line, and a unique identifier, the accession number, which is stored in the `AC` line. The content of most line types follows fixed rules or controlled vocabularies. This is very important for all searches based on text matching, and it is absolutely crucial for an automated treatment of data in TrEMBL.

2.2 TrEMBL vs. SWISS-PROT

Today, most protein sequences first appear as direct translations from nucleotide sequences which are produced in large-scale genome projects and are available through the DDBJ/EMBL/Genbank nucleotide sequence databases. All coding sequences in the nucleotide sequence databases are first translated into preliminary TREMBL protein sequence entries. These are checked for redundancy against existing SWISS-PROT and TrEMBL entries. Novel sequences are then annotated automatically and stored in TrEMBL. In addition to the sequence other information from the original EMBL entry is adopted as well, such as the source, organism, strain or tissue-type of the data, and references to publications and other databases. Some features can also be derived directly from the feature table of the EMBL entry, such as `signal`, `transit`, `chain`, `variant` and `conflict` [5].

TrEMBL entries are gradually moved into SWISS-PROT after manual curation by biologists. Thereby they are enriched with information extracted from publications or derived by sequence analysis. SWISS-PROT entries are regularly updated with data found in review articles or submitted by external experts. The production of TrEMBL is explained in detail in [1].

Due to the ever increasing output of data of sequencing centers, the time between the translation of a novel protein sequence from the nucleotide sequence and its annotation by a human editor is increasing. During this period, TrEMBL provides annotation which is achieved automatically by applying a set of analysis programs. Our goal is to retrieve as much information as possible with the highest possible level of quality.

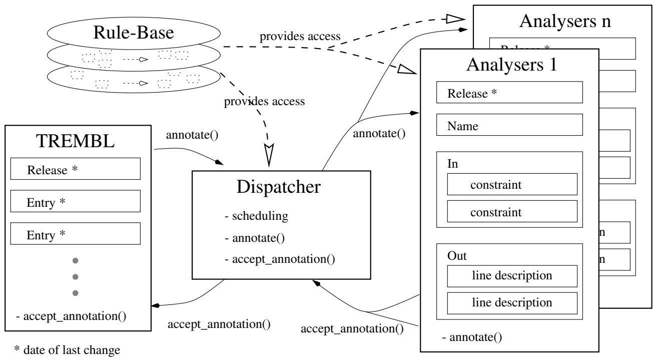

Figure 2: Participants in the Annotation System

3 EDITtoTrEMBL

The integration of programs can be implemented in multiple ways. Prior to the development of EDITtoTrEMBL the programs used for the automatic annotation of TrEMBL exchanged data via files and were controlled by a Makefile. This was sufficient as long as the interdependencies between programs were simple and could be modeled statically. EDITtoTrEMBL follows a different approach. It dynamically determines the procedure an individual sequence is undergoing based on a declarative description of the analysis programs. Thereby, different sequences are generally subject to different combinations of analysers in a different order. Details of the annotation process are discussed in section 4.

EDITtoTrEMBL is programmed in JAVA and uses the RMI mechanism (remote method invocation) for inter-process communication and distribution. Fig. 2 depicts the flow of data inside the framework.

3.1 Analysers

We use databases such as ENZYME ([4]), PROSITE ([3]), PFAM ([16]), and applications such as TMHMM ([17]) (prediction of transmembrane proteins) or NNPSL ([13]) (prediction of the sub-cellular location) as potential sources of protein annotation. Although there is a certain difference between these two methods, since databases are queried while applications are started, the system does not distinguish between them. In both cases we need to provide wrappers written in JAVA to support the physical distribution of annotation processes. These wrappers solve three tasks:

- Reformatting of a TrEMBL entry to a valid input for a program or a query. For programs, this is usually easy since most programs either accept TrEMBL entries directly or use FASTA format. For queries, the wrapper extracts certain parts of the TrEMBL entry which is then send to the database.
- Setting of parameters: Each wrapper chooses an optimal setting of parameters for each individual entry.
- Output rephrasing: To ensure consistency with the controlled vocabulary of SWISS-PROT , the program output is transformed according to a manually curated set of rules.

In the following, we call the unit of a wrapper with its associated program or database query an *analyser*. From the outside an analyser can be regarded as a black box which is fed with entries and returns them with additional annotation.

3.2 Interdependencies

Analysers are often highly specific. The correctness of their results depends partially on certain conditions, such as the taxonomic specification or certain keywords. Annotation added by an analyser is often in turn exploited by other analysers executed later.

Such analyser interdependencies can be rather simple or fairly complex. An example for the importance of the order in which analysers are called is NNPSL, which is used to predict the subcellular location of a protein. Before starting NNPSL it is necessary to assure that the protein is not a transmembrane protein. Hence the prediction of transmembrane proteins needs to be started before NNPSL is executed.

Another dependency occurs if certain analysers are known to give a more specific annotation than others, but are less sensitive. In this case it is advantageous to call less specific analysers only if the more sensitive analyser called earlier has failed to provide sufficient annotation.

Analysers can have cyclic interdependencies. For instance, the annotation that can be added following the identification of certain PROSITE patterns depends on the organelle where the protein is located. However, to predict this subcellular location using NNPSL, it first has to be assured that the protein is not transmembrane. This in turn can in some cases be achieved by identifying certain PROSITE patterns. Hence it is reasonable to start first PROSITE, then NNPSL, and then PROSITE again, assuming that a) the protein was found to be not transmembrane and that b) NNPSL could infer the organelle more precisely than known before.

It is clear that such situations are beyond the scope of the Makefile approach. EDITtoTrEMBL instead uses high-level descriptions of pre-conditions for the execution of analysers. These conditions are evaluated by *dispatchers*.

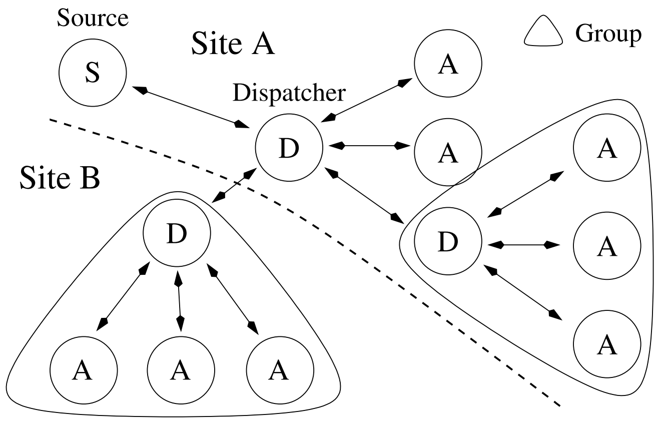

Figure 3: Analysers organized in a tree structure

3.3 Dispatcher

Dispatchers are programs that coordinate the flow of entries between different analysers. Whenever a new analyser is introduced into the framework, it is registered with a dispatcher. The dispatcher stores the name of the analyser together with a description of its pre-conditions and potential output. These are later used to determine dynamically the execution order of analysers for each entry.

Dispatchers can use other dispatchers to delegate tasks, which means that dispatchers can also act as analysers (see Fig. 3). This has several advantages:

- *Better maintenance*: The annotation process is broken into smaller subtasks which are easier to administer. We envisage that specialized sites will arise that treat certain aspects of the annotation autonomously.

- *Higher efficiency*: In general, a dispatcher will send an entry to each analyser whose pre-conditions are fulfilled. If the dispatcher and the analysers reside on different hosts, this will create significant network traffic. In contrast, if a dedicated dispatcher for a group of analysers residing on one host is installed on the same machine, an entry needs to be send via the network only once for the whole group.

3.4 Descriptions

Each analyser is characterized by its pre-conditions, the lines of an entry it uses for analysis and the potential result of its execution.

Pre-conditions are defined by two sets. Both sets consist of pairs comprised of a line tag and a regular expression. Each pair is interpreted as a condition which is fulfilled by an entry if the corresponding lines match the regular expression. The meaning of the two sets is the following: all conditions of the first set are *mandatory* pre-conditions. Their conjunction must be fulfilled by an entry before the analyser can be executed. Note that each condition can individually contain negation or disjunction. The second set describes which data is used by the analyser for its work, the *optional* constraints. Lines of an entry which match the optional constraints of an analyser are called *active lines*.

The biological meaning of the conditions can be rather technical, such as stating that a protein must have an enzyme classification number to be treated as an enzyme. It may also carry more semantics, such as expressing that the entry must not describe a transmembrane protein.

The potential result of the analysis is described in a similar fashion. It is not used during workflow planning but considered by the heuristics which select an analyser in case that many are possible. For this purpose, we need to match pre-conditions with the potential outcome of analysers to make a good "guess". We describe the output as a set of pairs, where each pair consists of a line tag and a text. This text could be a list of possibly resulting keywords. As it is in general not possible to specify all potential results, we sometimes need to fine-tune the output descriptions with respect to existing pre-conditions.

The description of a dispatcher is composed of the description of its registered analysers: input descriptions are logically connected by disjunction, whereas the output description is the union of all output descriptions of analysers.

3.5 Workflow Planning

Upon retrieving an entry for annotation, a dispatcher proceeds as follows. If first determines all *active analysers*. We call an analyser active if two conditions are met: first, its pre-conditions must be fulfilled. Second, there must have been a change in one of its active line tags after its last execution. If it was not executed before, the second condition is obsolete. The dispatcher selects one of the active analysers, using a heuristic explained below, and executes it. It repeats this procedure until no more analysers are active or the annotation has reached a certain size.

The intuition behind this algorithm is the following. In general, analysers work the better the more precise knowledge they have. At the beginning only little knowledge can be directly derived from the entry's annotation, and hence analysers will necessarily make vague and conservative decisions. During the process more and more knowledge is accumulated, and often analysers should be called again to make better annotation based on new evidence.

3.6 Environment

Our system follows a client-server architecture. Clients request annotation by sending an entry, or a set of entries, to a root dispatcher. The entire procedure of analyser selection and process distribution is completely transparent for the client. A dispatcher uses its registered analysers for annotation, but can also create further instances if necessary by starting remote shells.

Analysers, respectively their wrappers, are in general started and stopped independently from the dispatcher. Upon startup they first register with a dispatcher and then persist in memory, waiting for entries. This does not imply that the actual analyzing program itself also stays in memory; this depends on the implementation of the program, which cannot be influenced by the framework, and is independent of the wrapper.

We currently have implemented two clients. One for the annotation of the entire TrEMBL database; a second provides a graphical user interface for the annotation of single entries.

3.7 Distribution

The distribution of the annotation process could also be achieved with standard Makefiles using a queuing system. EDITtoTrEMBL conceptually expands the scope of distribution to every remote host connected via the Internet. This is possible due to the usage of JAVA's RMI mechanism.

Load balancing is performed inside every dispatcher. Naturally, it applies only if several instances of one analyser are available. An entry is then sent to that instance with the least number of entries in procession. The dispatcher might also decide to recruit more analysers if possible.

4 The Annotation Process

In this section we explain the annotation process in detail. We first introduce a formal description and then explain the algorithm. Finally we highlight possible pitfalls and how we avoid them.

Let A be the set of analysers. Let e be an entry, E the set of all possible entries. Let $F_a(e) : E \mapsto E$ be the function that transforms an entry into another entry by applying analyser a. We call $(a_1, \ldots, a_k)$ with $a_i \in A, 1 \leq i \leq k$ a *path*; it describes an ordered series of analyser executions. The annotation achieved by this path is $F_{a_1,\ldots,a_k} := F_{a_k} \circ F_{a_{k-1}} \circ \ldots \circ F_{a_1}$. Note that analysers can appear more than once in a path.

For every analyser $a \in A$ we define the preconditions as a function $C_a : E \mapsto \{true, false\}$. C_a is evaluated on an entry e by testing all mandatory constraints and returns *true* if they all evaluate to true. $C_a^+ : E \mapsto \{true, false\}$ is a second function, built from the optional constraints. C_a^+ returns true if *any* of the optional constraints evaluates to true. O_a is the description of the potential output of a. An analyser a is completely characterized by the triple $\{C_a, C_a^+, O_a\}$.

We also introduce a subset relation on entries. $e_i \leq e_j; \quad e_i, e_j \in E$ means that e_i has a subset of the annotation of e_j. With $e^{C_a^+}$ we denote the entry e restricted to those lines contained in C_a^+.

We do not allow analysers to remove any line of annotation. Hence, analysers are *monotonous* in the sense that $e \leq F_a(e)\ \forall a \in A, e \in E$.

4.1 Algorithm

Our algorithm for incremental annotation of a single entry proceeds as follows (Figure 4). If a client wants an entry e to be annotated it will call the function *annotate* of the root dispatcher D. A is the set of analysers known to D. D then basically performs a loop. In each iteration first all active analysers are computed (function *active*). Of those, one is chosen using a heuristic (function *choose*, see below). The entry is annotated by this analyser and the loop starts again. If no more annotation can be added, the entry undergoes a final postprocessing to reduce redundancy and increase consistency (*summarize*, see below).

An analyser a needs to fulfill two conditions to be considered as active for an entry e. First, all its pre-conditions must be true (function $C_a(e)$). Second, we check if the execution of a will produce any additional results. It is reasonable to presume that all analysers are idempotent,

```
fun annotate(e : E, T : N) : E
  var
    candidates : ℘(A) = ∅;
    history : array A → E;
    count : array A → N;
    e : E;
  begin
    foreach a ∈ A do
      history[a]:="";
      count[a]:= 0;
    od
    do
       candidates := call active(history, e);
       a := call choose(candidates);
       e := call F_a(e);
       if ¬call addhistory(history[a], e)
         then STOP;
       fi
       count[a] + +;
    od(#candidates = 0 ∨ count[a] = T)
    return call summarize(e);
  end
where
fun active(history : array A → E, e : E) : ℘(A)
  var
    diff : string := "";
    analysers : ℘(A) := ∅;
  begin
    foreach a ∈ A do
      if call C_a(e)
        then
             if history[a] = ""
               then
                    analysers+=a;
               else
                    diff := e − history[a];
                    if call changed(a, diff)
                      then
                           analysers+=a;
                    fi
             fi
      fi
    od
    return analysers;
  end
```

Figure 4: Algorithm for annotation. *summarize*, *choose*, *changed* and *addhistory* are explained in the text.

which means that always $F_a \circ F_a = F_a$. Hence, a should not be executed if nothing has changed in an entry it was applied to before. Using C_a^+, we can formulate more stringent conditions, demanding that something has changed in e at a positions that affects the analysis of a, that is there must have been a change in $e^{C_a^+}$.

To evaluate the second condition, D keeps a history list for every analyser, storing the result of its last application on e. *active* uses this to compute the difference between the current entry and the entry stored in the history. It then checks if the difference matches with one of the conditions of C_a^+ (function *changed*). The history is furthermore used to check the consistency of each analyser inside the function *addhistory* (see below).

Every single analyser in the system is assumed to work correctly, since the analyser was empirically verified on SWISS-PROT.

For sets of entries, such as a complete database, the algorithm is started once for every entry. Note that an execution of an analyser can actually mean that the entry is passed to another dispatcher which takes the role of an analyser.

4.2 Properties of the Algorithm

Termination: Using the monotonicity stipulated in the previous section, we can follow: let $e_n := F_a(e_{n-1})$, $n \geq 1$ for some $a \in A$, e_0 being the un-annotated entry. Then $e_j \leq e_k, \forall j \leq k$. Hence, the total amount of annotation is never getting smaller, but either stays unchanged or grows with every iteration of the main loop in *annotate*.

In general, we observed that analysers have only a limited 'knowledge capacity'. Multiple revisions of annotation created earlier occurs rarely. After a few rounds the list of active analysers is empty and the algorithm terminates. However, one can imagine pathological cases in which for instance two analysers mutually add data into the others C^+-lines, provoking an infinite loop. To exclude this unlikely possibility we set on upper limit on the number of times an analyser may be executed (variable T).

In the following we assume that the algorithm stops naturally when all analysers have been fully exploited, which means that the annotation reaches a fixpoint.

Negative Pre-Conditions: Without allowing negative constraints the order in which analysers are executed would not matter. In this case, all possible paths would result in the same annotation - the annotation process has a unique fixpoint.

However, with negative constraints this does not hold any more. Following a different path will lead to different annotations. Consider the following example:
$a_1 := \{\{KW : \neg$``DNA-BINDING''$\}, \{\}, \{CC :$ ``SUBCELLULAR LOCATION''$\}\}$,
$a_2 := \{\{CC : \neg$``SUBCELLULAR LOCATION''$\}$, $\{\}, \{KW :$ ``DNA-BINDING''$\}\}$.

Calling a_1 first will inhibit the execution of a_2 and vice versa. Both paths are possible, and they lead to different annotation. As the result of an actual execution of an analyser is not known in advance, we cannot judge which of the two paths is better. Note that this problem is closely related to the problem of negation in deductive databases (see for instance [18]).

Heuristics: In the last section we argued that we cannot make a decision which of the possible paths will result in the best annotation. We therefore use a *heuristic* (function *choose*) to select one of the active analysers at each iteration of *annotate*. Several strategies are possible:

1. Prefer analysers with tight constraints since they are highly specific and use more knowledge.
2. Prefer analysers with loose constraints since those are more general and should be executable with minimal influence on other analysers.
3. Try to prevent changes to lines that could block other analysers having negative constraints as long as possible.
4. Call analysers first that make changes to lines on which other analysers have negative constraints to prevent potentially wrong annotation.
5. Use additional priority information specified by the administrator of the system.

Currently we experiment with combinations and variants of those heuristics to find out which yields the best result.

Summarization: A final processing of an entry has two goals: first, it tries to reduce redundancy. Second, it spots possible inconsistencies.

Regarding consistency, we require each analyser to be *consistent with his own, previously drawn decisions*. To test this, we check whether the new annotation of an analyser contradicts its previous (function *addhistory*). If this happens, we stop the process, assuming that the annotation is erroneous. In such cases the specific analyser and its description need to be carefully checked by the administrator.

Note that this only assures the consistency of a single analyser with itself. Checking consistency between the annotations made by different analysers is more difficult and subject to further research.

Regarding redundancy, we follow a similar approach. All the annotation is temporarily stored in a hidden section of the TrEMBL entry. As the last step of the algorithm, all annotation made by one specific analyser is deleted except the last, which is then shifted into the visible section of the entry. The above described consistency check assures that we do not loose any information through this clean-up. Checking redundancy between different analysers is not yet tackled.

4.3 Example

Figure 5 visualizes a simple example. It contains two analysers, $A = \{\{\texttt{SQ, DR}\}, \{\}, \{\texttt{CC}\}\}$ and $B = \{\{\texttt{SQ}\}, \{\}, \{\texttt{DR}\}\}$ Imagine that B is an analyser that searches for PROSITE patterns, therefore needing only the protein sequence, and A finds an annotation for the comment lines ($O_A = \{\texttt{CC}\}$) according to these patterns. If B finds a pattern, it will add this to the description lines of the entry ($O_B = \{\texttt{DR}\}$).

The output descriptions in this example are good enough since the constraints do not demand more detailed information. There is no need for the descriptions to be more detailed than the constraints, except for easier maintenance. In general, output descriptions are more specific, such as {"DR PROSITE", "KW TRANSMEMBRANE DNA-BINDING"}. In this example, we assume that the output is actually generated. We do not use the set of optional constraints.

Each group of line tags in figure 5 symbolizes an entry, respectively its fulfilled constraints. The regions separate *realms* of analysers. Those with a $*$ *before* the analyser name contain all entries that fulfill all pre-conditions of the analyser (the *before region*, $*a$); those with a $*$ *after* the name represent all entries that fulfill both the pre-conditions and contain the annotation that results from the execution of the analyser (the *after region*, $a*$).

The highlighted entries mark the (partial) path of an entry through the annotation. Initially, only B can be applied, as the DR precondition of A is not fulfilled. Executing B adds a DR line, moving the entry from $*B$ into $B*$.

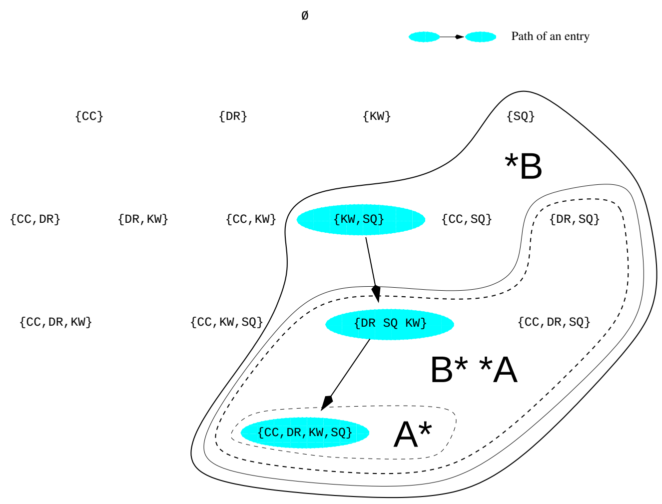

Figure 5: Visualization of Analysers

It is now also in $*A$, which is executed next. A adds a CC line, moving the entry into $A*$.

An entry in general lies in the *before region* of more than one analyser. These together form the set of active analysers (see 4.1) for this entry. With the executing of an analyser a the entry jumps from $*a$ into $a*$, possibly entering *before regions* of new annotators. Such a move is performed in every iteration of the algorithm, assuming that a was successful. Note that it is not possible to compute the actual path in advance, since output descriptions only give *potential* results. It is not known which of these results will emerge from an execution on an arbitrary entry.

Such a visualization helps to understand the dedication of analysers to different problem domains and to get a general impression of their performance.

5 Discussion

In this article we presented EDITtoTrEMBL as a new approach to the automatic annotation of protein sequences. EDITtoTrEMBL is a step forward to fulfill the requirements for sequence annotation frameworks postulated in the introduction. We ensure biological correctness by having each analyser being checked by an expert in the field. By using JAVA wrappers we achieve language and platform independence and can use RMI for a comfortable implementation of process communication. Dispatchers automatically distribute processes using a simple load-balancing algorithm.

One of the main differences to other approaches to automatic annotation, such as PEDANT ([8]), GeneQuiz ([7]) or MAGPIE ([10]), is the dynamic, data-driven workflow planning. It allows us to treat every single entry in a way exactly tailored to its data. This saves time, as unnecessary processes are avoided, and enhances the quality of the annotation. While traditional approaches tend to use only a few means of annotation on all entries, we rather try to integrate many different programs and let them work only on those entries for which they are best suited and manually proven to be correct.

We, and other groups, see it as advantageous to separate the sequence annotation into distinct *modules*. The most important is a stable framework module, responsible for communication, process planning and intelligent combination of results. For instance, the GAIA project ([2]) handles communication through a relational database serving as a black board. Analysers, called *sensors* in their environment, write results into that database. Later, other modules read from the database, trying to integrate results. GAIA does not include any planning facilities but simply gives each entry once to each sensor.

Both GAIA and Genotator ([12]) have a strong focus on the visualization of annotation information. We do not treat this issue in this paper, because we believe that visualization is another module whose development is orthogonal to that of the framework.

Our current implementation is already nearly as fast as the static Makefile-based approach. Considering the increased communication effort and the fact that there are currently only a few analysers installed, we found this result very encouraging. With the number of analysers increasing, the balance between communication overhead and time-saving through more intelligent distribution and planning will certainly be favorable.

We also plan further enhancements. To reduce communication time, we will pass the task of query planning to the entry itself, eliminating the need to permanently revisit dispatchers. We are also investigating other methods for formulating pre-conditions and output descriptions, being more expressive than the currently used regular expressions. Furthermore, the planning algorithm itself is subject to improvements that will increase the degree of parallelization.

EDITtoTrEMBL allows us to focus on the development of new analysers and the evaluation and integration of existing programs, respectively. We particularly envisage to extend our fruitful collaboration with on-line experts for SWISS-PROT to the generation of on-line programs for TrEMBL.

Acknowledgment

This work was in part supported by EMBO fellowship no. ALTF ASTF, the German Science Foundation, grant no. GRK 316, and the European Commission, grant no. BIO4-CT97-2099.

References

[1] Rolf Apweiler, Claire O'Donovan, Maria Jesus Martin, Wolfgang Fleischmann, Henning Hermjakob, Steffen Moeller, Sergio Contrino, and Vivian Junker. Swiss-prot and its computer-annotated supplement trembl: How to produce high-quality au-

tomatic annotation. In *World Multiconference on Systems, Cybernetics and Informatics*, Orlando, Florida, 1998.

[2] Charles Bailey, Stephen Fischer, Jonathan Schug, Jonathan Crabtree, Mark Gibson, and G. Christian Overton. Gaia: Framework annotation of genomic sequences. *Genome Research*, 8:234 – 250, 1998.

[3] A. Bairoch, P. Bucher, and K. Hofmann. The PROSITE database, its status in 1997. *Nucleic Acids Research*, 25:217–221, 1997.

[4] Amos Bairoch. The ENZYME data bank in 1995. *Nucleic Acids Research*, 24:221–222, 1996.

[5] Amos Bairoch. SWISS-PROT user manual. http://www.expasy.ch/txt/userman.txt, 11 1997.

[6] Amos Bairoch and Rolf Apweiler. The SWISS-PROT protein sequence data bank and its suplement TrEMBL. *Nucleic Acids Research*, 26(1):38–42, 1998.

[7] G. Casari, Christos Ouzounis, Alfonso Valencia, and Chris Sander. Genequiz II: automatic function assignment for genome sequence analysis. In *Proceedings of the First Annual Pacific Symposium on Biocomputing*, pages 707–709. World Scientific, 1996.

[8] D. Frishman and H.-W. Mewes. Protein structural classes in five complete genomes. *Nature Structural Biology*, 4:626–628, 1997.

[9] Terry Gaasterland, Peter Karp, Kevin Karplus, Christos Ouzounis, Chris Sander, and Alfonso Valencia, editors. *Proceedings of the 5th International Conference on Intelligent Systems for Molecular Biology.* AAAI Press, Menlo Park, CA, 1997.

[10] Terry Gaasterland and Christoph Sensen. Magpie - automated genome interpretation. *Trends in Genetics*, 12(2):76 – 78, 1996.

[11] D. Georgskopoulos, M. Hornick, and Amit Sheth. An overview of workflow management: From process modeling to workflow automation infrastructure. *Journal of Distributed and Parallel Databases*, 3(22):119 – 154, 1995.

[12] N. L. Harris. Genotator: A workbench for sequence annotation. *Genome Research*, 7:754 – 762, 1997.

[13] Astrid Reinhardt and Tim Hubbard. Using neural networks for prediction of the subcellular location of proteins. *Nucleic Acid Research*, 26(9):2230–2236, 1998.

[14] Genome Research, editor. *Special Issue on Bioinformatics*, volume (8):3. Cold Spring Harbor Laboratory Press, 1998.

[15] Patricia Rodriguez-Tome, Peter J. Stoehr, and Graham N. Cameron. The European Bioinformatics Institute (EBI) databases. *Nucleic Acids Research*, 1997.

[16] Erik L. L. Sonnhammer, Sean R. Eddy, Ewan Birney, Alex Bateman, and Richard Durbin. Pfam: Multiple sequence alignments and HMM-profiles of protein. *Nucleic Acids Research*, 26:320–322, 1998.

[17] Erik L. L. Sonnhammer, Gunnar von Heijne, and Anders Krogh. A hidden markov model for predicting transmembrane helices in protein sequences. In *Proc. of Sixth Int. Conf. on Intelligent Systems for Molecular Biology*, Menlo Park, CA, 6 1998. AAAI Press.

[18] Jeffrey D. Ullman. *Priciples of database and knowledgebase systems*, volume one of *Principles of computer science series*. Computer Science Press, 1988. ISBN 0-7167-8158-1.