Logol: Expressive Pattern Matching in Sequences. Application to Ribosomal Frameshift Modeling

Catherine Belleannée, Olivier Sallou, and Jacques Nicolas

Irisa/Inria/Université de Rennes1, Campus de Beaulieu,
35042 Rennes, France
{Catherine.Belleannee,Olivier.Sallou,
Jacques.Nicolas}@irisa.fr

Abstract. Most of the current practice of pattern matching tools is oriented towards finding efficient ways to compare sequences. This is useful but insufficient: as the knowledge and understanding of some functional or structural aspects of living systems improve, analysts in molecular biology progressively shift from mere classification tasks to modeling tasks. People need to be able to express global sequence architectures and check various hypotheses on the way their sequences are structured. It appears necessary to offer generic tools for this task, allowing to build more expressive models of biological sequence families, on the basis of their content and structure.

This article introduces Logol, a new application designed to achieve pattern matching in possibly large sequences with customized biological patterns. Logol consists in both a language for describing patterns, and the associated parser for effective pattern search in sequences (RNA, DNA or protein) with such patterns. The Logol language, based on an high level grammatical formalism, allows to express flexible patterns (with mispairings and indels) composed of both sequential elements (such as motifs) and structural elements (such as repeats or pseudoknots). Its expressive power is presented through an application using the main components of the language : the identification of -1 programmed ribosomal frameshifting (PRF) events in messenger RNA sequences.

Logol allows the design of sophisticated patterns, and their search in large nucleic or amino acid sequences. It is available on the GenOuest bioinformatics platform at *http://logol.genouest.org*. The core application is a command-line application, available for different operating systems. The Logol suite also includes interfaces, e.g. an interface for graphically drawing the pattern.

1 Background

During last decade, a number of pattern matching tools have been proposed, and some of them are used extensively and helpful. Depending on systems under study, modeling needs may vary from looking for all exact occurrences of a given string in a protein bank, to looking for approximated occurrences of a

M. Comin et al. (Eds.): PRIB 2014, LNBI 8626, pp. 34–47, 2014.

given transposon in a full genome, or locating pseudoknots in a RNA sequence. This section proposes a quick overview of the existing software diversity, which shows there exists still room for a new pattern matching tool, both flexible (high "genericity") and with the capacity to represent complex structures (high expressive power).

1.1 General Purpose Pattern Matching

Some tools have been designed for the analysis of several types of sequences (DNA, RNA, proteins) with an generic expressiveness, i.e. without targeting the recognition of a particular motif family. Among these general tools, two tendencies can be observed, efficiency-oriented and expressiveness-oriented software.

One of the most advanced software from the point of view of efficiency is the Vmatch suite (`http://www.vmatch.de`) that offers a large variety of search facilities in very large sequences. It is based on a careful implementation of enhanced suffix trees for the computation of a sequence index that provides a fast access to every substring in that sequence. If the search for a motif contains some rare substrings, this technique is particularly efficient. The software Vmatch is the core search engine used in a number of more specialized tools working on specific sequence structures (e.g. "tandem-repeats" or LTR retrotransposons). Another highly generic tool is Biogrep[11], designed with the objective of *quickly recognizing a large set of simple motifs* (typically more than 100) in biological sequence banks. Biogrep allows queries in the POSIX language, a standard format of extended regular expressions, and can look for patterns in parallel on a set of processors.

The other approach for the analysis of biological sequences is more concerned with modeling the peculiarities of biological objects in the most relevant and expressive way. A major contribution in this respect is the work of D. Searls who laid the foundations for the research in this domain. He was the first to supervise developments allowing users to design biological grammars and to apply them for the large scale analysis of their genomic sequences [19,4,18]. D. Searls has introduced a very practical object in algebraic grammars, the *string variable*, which allows to elegantly express the notion of copy (either direct or reverse). He has implemented the resulting logical formalism, called SVG -for String Variable Grammars-, in the (no longer available) Genlang tool [4]. The direct copy (*e.g.* `X...X`) allows to search for two occurrences of a same unknown string, using optionally some indication on the string size. The reverse copy (*e.g.* `X... ~X`) introduces in addition the notion of reverse complement and allows thus the representation of biological palindromes like stem-loops (`Stem, Loop, ~Stem`) or pseudo-knots (`Stem1, Loop1,Stem2, Loop2, ~Stem1, Loop3,~Stem2`). Genlang, Stan[15] (developed in our research team), Patscan[5] and Patsearch[16] are all tools belonging to this family. Thanks to string variables and other additional components, these languages offer the possibility to combine easily in a single model informations on the sequence and on the structure of a molecule.

1.2 Dedicated Pattern Matching

It is not possible to provide here an exhaustive review of the profusion of specific tools that have been made available to bioanalysts. Some are specific to a sequence family and others to a particular motif type. A famous one *dedicated to proteins* is ScanProsite[3], where motifs are built upon regular expressions that are searched either by a query in a precomputed database or with the algorithm PS-SCAN[9]. A number of tools are *dedicated to RNA sequences*, in response to the increasing needs of structure exploration in the complex RNA world boosted by the recent importance of non coding RNA studies. For instance, RNAmotif[13], RNAbob[6], Hypasearch[10,20] and Palingol[2] have been designed for the description of patterns as a succession of stems and loops, usually offering the possibility of choosing either a standard Watson-Crick pairing (`A-U`, `G-C`) or a pairing including Wobble (`A-U`, `G-C`, `G-U`). A more recent tool in this category, Structator[14], significantly improves the parsing time by making use of an index structure that is suited for the analysis of palindromic structures, the affix arrays. Patterns may also contain some sequence information on words that have to be present in particular places of the stems or the loops. RNAmotif is probablly the most popular in this category.

2 Logol Language

In this landscape, we designed Logol, a new general grammatical pattern matching tool, in order to greatly enhance the range of admissible patterns.

2.1 Basics: A Grammatical Model with Constraints

• **String Variable Grammars:** With the objective of being a general and expressive language allowing a natural expression of composite patterns, the Logol language has been designed on the basis of String Variable Grammars (SVG) introduced by D.Searls [19,4,18]. As already mentioned in the previous section, while it is easy to express motifs and gaps by regular expressions (*e.g.* PROSITE [3]), SVG allow to express structures beyond the capabilities of regular languages such as palindromes (e.g. in stem-loops and pseudo-knots) and repeats (duplicated substrings), which are recorded by string variables. In fact, SVG and Logol grammars lay even beyond the possibilities of context-free grammars (XML-like), in a class that A. Joshi called "*mildly context sensitive languages*" [12]. Starting from the sound basis of SVG grammars, the Logol language proposes several extensions -most notably by adopting a constraint approach- with the goal to allow the expression of realistic biological motifs. The rest of the section introduces its main constituents.

• **First Steps in Logol:** Let us first present a very simple Logol grammar:

```
mod1()==*> SEQ1
mod1()==>"aaa"
```

The first line is the top level instruction (called 'rule'). The rule is identified by the constant '`==*> SEQ1`' and it triggers the parsing of a sequence for a particular grammatical model (here, `mod1`). The second line provides the model (i.e. pattern) definition itself, here the string `"aaa"`. It triggers the search for all occurrences of "*aaa*" in a genomic sequence.

The next grammar describes a slightly more interesting pattern made of two distant copies of a same string. The size of the string (in range [5,8]) and the distance between the two copies (in range[1,10]) are bounded but the content of the string itself is kept free:

```
mod2()==*> SEQ1
mod2() ==> X1:#[5,8], .*:{#[1,10]}, X1
```

The model `mod2()` reads as follows: `X1` denotes a string variable; any string made of 5 to 8 letters can be an instance of `X1`. '`.*`' denotes a space ('gap'). It is constrained to have a size between 1 and 10 characters. A second occurrence of `X1` is waited for after the gap. For example, **acuggc***ccgacuggc***acuggc** is an instance of this pattern on the input sequence *uucagacuggcccgacuggcacuggccac*, with `X1` = *acuggc*.

• **All Matches:** Logol returns all the instances of a model. For instance, when launched on the sequence *cagaaaacgccgaaacuggc* with the model `"aaa"`, it returns the three possible matches: in position 3, 4 and 12.

• **A Powerful Feature: Instance Saving.** The Logol language supports an alternative way to express the pattern `mod2`. That is:

```
mod2() ==> X1:{#[5,8],_IX1}, .*:{#[1,10]}, ?IX1.
```

In this case, the string corresponding to the occurrence value of `X1` is saved (using '`_`') in a new variable (named here `IX1`). After a gap of length 1 to 10, the same string `IX1` is required again and called back using '`?`'.

This complicated version of `mod2()` is only shown for the purpose of introducing the notion of *instance saving* that will be fully used in the next paragraphs. Actually, the various instances of a variable are not necessarily exact copies and this explicit naming process (here `_IX1`) makes it possible to distinguish one instance from another. Furthermore, such a mechanism allows to save some instance in any part of a model and refer to it elsewhere in the model.

2.2 Constraints

Logol modeling is based on a *constraint approach*. The various constraint types applicable to a model element may be split into two categories, *string constraints* and *structure constraints*. String constraints delimit the start (`@`), the end (`@@`), the content (`?`) and the length (`#`) of admissible strings. Structure constraints include cost constraints (`$` for mismatch count, `$$` for indel count) and composition constraints (`%`). These two categories of constraints are written in two separated sets, as in the following model: `mod1() ==> X1:{#[6,7],@[3,11]}:{% "a":50}`. Here `mod1` looks for a string whose size is in range `[6,7]`, which starts at a

position in range `[3,11]` and contains at least 50% of `a`. For example, the instances of this pattern in the sequence *ccaaaacgtacgtttttttccccccc* are *aaacgt*, *aaacgta* and *aacgta* (positions start at 0 in a sequence).

• **Non Exact Copies: Mismatch and Indel Cost Constraints (`$` and `$$`)** Genomic sequences evolve through a duplication process prone to errors or mutations. Elementary variations (on one position) between a model and its instances are taken into account through two dedicated cost counters: the counter of *mismatches* (i.e. substitutions) and the counter of *indels*. A mismatch cost constraint is defined by a `$[m,n]` expression, where m and n are integers. This constraint allows from m to n substitutions. For example, *aaaa*, *acaa* and *aagt* are all instances of the pattern `"aaaa":{$[0,2]}`.

A mismatch constraint can also take the form of a rate: `p$[m,n]`. Here, m (resp. n) designates the minimum (resp. maximum) allowed percentage of substitutions. For example, *aaaa*, *acaa* and *aagt* are all instances of the pattern `"aaaa":{p$[0,50]}`. Indels are defined similarly, by setting the indel cost constraints `$$[m,n]` and `p$$[m,n]`. Thus, `"aaaa":{$$[0,1]}` is accepting, among others, the strings *aaaa* (no indel), *aaa* (one deletion) or *aaaca* (one insertion). Here is a new example to further illustrate the concept of instance saving:

```
X1:{#[5,8],_I1},.*:{#[1,7]}, ?I1:{_I2}:{$[1,1]}, .*:{#[1,7]}, ?I2:{$[1,1]}
```

This model allows to look for 3 instances of a same string successively deriving from each other (e.g `I1` =`aaaaa`, `I2` = `aaaca` and `I3` = `agaca`). The second pattern, `?I1: {_I2}:{$[1,1]}`, reads as follow: the expected string must be similar to the previous `I1` string (`aaaaa` here), apart from 1 mismatch (`$[1,1]`). The matched string (`aaaca`) is saved in `I2` (`{_I2}`) for further use (`{?I2}`). This individualization of instances allows to adjust fine notions of sequence evolution.

• **Letter Frequencies: Composition Constraints (`%`)**: Some properties like hydrophobic regions in proteins or GC content in RNA correspond to statistical expectations on a particular segment composition rather than the search of a well-defined element. Logol proposes the expression of *composition constraints* that check the relative frequency of given letters in a sequence. Thus `X1:{#[2,43]}:{% "gc":65}` describes a segment of length 2 to 43 characters with a GC rate of at least 65%.

2.3 Operators

• **Negation:** Also called *negative content constraint*, negation can be used in order to exclude some values in a motif. It is denoted by the exclamation mark symbol, `!`. Thus `("aaa"| "ttt"), !"ga":{#[2,2]}` refers to a string made of 5 characters, the first three being 3 `a` or 3 `t`, and the next two being anything but the word `ga`.

• **Morphism:** A morphism is a function that applies a transformation to a string by substituting letters or substrings. It can be used in direct (`+`) or reverse (`-`) direction. Each user can define its own morphisms, but some are already defined. For instance, `"wc"` transforms a RNA sequence into its complement

sequence, applying the Watson-Crick pairing (`A-U`, `G-C`). Thus, the pattern `+"wc" "acuggc"` represents the string `"ugaccg"` and `-"wc" "acuggc"` represents the string `"gccagu"`.

The morphism `-"wc"` produces the *reverse complement* of a string and can be used to describe biological palindromes such as stem-loops. The next example provides a pattern for the recognition of stem-loops whose stem length varies between 5 and 11 and loop size between 1 and 9. Moreover, the Watson-Crick pairing is not required to be perfect: up to 2 substitutions and 1 indel are allowed.
`STEM1:{#[5,11],_IS1}, .*:{#[1,9]}, -"wc" ?IS1 :{$[0,2],$$[0,1]}`
In this description, the content of `STEM1` (first strand of the stem) is saved in `IS1`, (`_IS1`). The second stem strand is then defined as the exact reverse complement of the previous content (that is `-"wc" ?IS1`), except for 2 mismatch and 1 indel.

• **Repeats:** Tandem repeats are frequent genomic structures made of directly adjacent copies of a same entity that may contain only a few letters (microsatellites) or be longer and reach size over 100 nucleic acids (minisatellites). In Logol, such structures can be handled by applying a special constructor, `repeat`, which manages the characteristics of series of occurrences. Its standard format is:
`repeat(<entity>,<distance>)+<occurrence number>`. For instance,
`repeat("acgt",[0,3])+[7,38]` states that substring *acgt* is repeated from 7 to 38 times, using a spacing of at most 3 characters between 2 repeats.

• **Views and Scope of Constraints:** Constraints (on the content, the size...) can be set on various parts of a model. They can be imposed to elementary entities like strings or variables as it has been shown previously, or to a set of entities that have themselves individually their own constraints.

If the set represents *contiguous elements*, it is called *a view*. In Logol syntax, a view is delimited by parentheses. In the following example, the model considers strings built from the concatenation of the instances of 3 variables `X1`, `X2` and `X3`, each one having length up to 10 characters. A supplementary constraint on the view made of the whole string `(X1,X2,X3)` requires that its total length is bounded between 8 and 20 characters.

```
(X1:{#[1,10]}, X2:{#[1,10]}, X3:{#[1,10]} ) : {#[8,20]}
```

It is also possible to set some constraints on a collection of *non-contiguous elements* (for instance on the two segments that form the stem of a stem-loop in a RNA structure). Such constraints are set in this case in a specific global module, the *control panel*. The following example details a stem-loop structure made of two stem elements that have to contain globally at least 30% of C.

```
controls:{
% "c"[mod1.ISTEM1,mod1.ISTEM2]>=30
}
mod1()==> STEM1:{#[2,18],_ISTEM1},.*:{#[1,10]}, -"wc" ?ISTEM1:{_ISTEM2}
mod1()==*>SEQ1
```

• **Multiple Analyses:** The coexistence of alternative structures in a same region is certainly amongst the important features of biological sequences. Gene overlapping for instance has been found in all kingdoms of life, including viruses and

higher eukaryotes. Logol allows to model such situations by stating alternative models in the grammar top rule (`==*> SEQ1`). Then, a sequence is accepted only if it contains an instance of each possible alternative. For instance the grammar:

```
mod1().mod2()==*> SEQ1
mod1() ==> "yvcpfdgcnk"
mod2() ==> "nklkshil"
```

accepts the sequences containing both the strings *yvcpfdgcnk* and *nklkshil*, independently of their positions, being overlapping or not. Parameter-passing is possible between alternative models, e.g. to settle the respective positions of alternative elements.

```
mod1(SAVE1).mod2(SAVE1)==*> SEQ1
mod1(SAVE1) ==> "aata":{_SAVE1},X1:{#[30,30]}:{% "gc":60}
mod2(SAVE1) ==> "gggcaa":{@[@SAVE1 - 20,@SAVE1 + 20]}
```

The above model is looking for an instance of string *aata* that is both followed by a GC-reach area and contains a neighboring occurrence of string *gggcaa*. Indeed, `@[@SAVE1 - 20,@SAVE1 + 20]` constrains the *gggcaa* string to be located 20 nt before or after the *aata* string.

3 Logol Implementation

3.1 Input /Output Specifications

The Logol software is in charge of matching a Logol pattern against one or more (DNA, RNA or protein) sequences, in order to point out all the pattern occurrences within the sequences.

To this end, it needs two main inputs: a Fasta file with the sequences to be analyzed, and a textual file with the grammatical rules of the pattern. The tool accepts also a configuration file, setting some parsing parameters. This allows, among other, to limit the scope of the search by limiting the maximum number of matches, to choose the indexing tool (Vmatch or Cassiopee, see below), or to detect and filter irrelevant match variants.

The application outputs a zip archive containing one XML file per input sequence. Output files record all the details of the matches, including the matching rule, the location of the match, its length, and the number of substitutions and indel. It also keeps the match information in a tree hierarchy. A repeat, for example, will be decomposed in an array of matches. Thus, it is possible to analyze the result of the global match, but also the details of any element of the grammar. In addition to the XML output, with match details, the analyzer can also output the results in Fasta or GFF3 format. Those formats ease exploiting results within workflows using other tools.

The core application to launch a Logol analysis is a command-line application, available for different operating systems. However, there exists also user interfaces, for more comfort. Among them, the *model designer* let the user draw the model graphically, and the web application converts it into a Logol grammar.

3.2 Sequence Analysis

Pattern matching is performed in two stages. At first the Logol pattern is deciphered by the grammar analyzer, then it is applied on the input sequences by the sequence matcher.

• **Parser:** The grammar analyzer is a Java program. Its role is to decode the grammar to generate a script used by the sequence matcher, and to launch the calls to the sequence matcher. The generated script is a Prolog file which uses a dedicated library containing Prolog predicates for each kind of grammar element (spacers, repeats, ...). The grammar follows a DSL (Domain Specific Language) analyzed by the Antlr library (`http://www.antlr.org/`). The Prolog programming language has been chosen for its flexibility and conciseness in expressing parsers, due to its built-in ability for backtracking on partial solutions and natural handling of non determinism. However, the implementation could have been achieved with any other language.

To generate the script, several parsing runs are achieved. The first parsing stage gathers information on each element (expected position range, minimum and maximum size, number of allowed errors, ...).

A second parsing stage tries to solve cases where a variable is used but will be instanciated later in the model, e.g in *"acgt",?X1,"cgta":{ _X1 }*. Indeed, though the grammar itself does not require a left to right reading, we use in practice a left to right parsing of the sequence. To manage such cases, the tool finds the variables in this specific situation and applies a dedicated search technique. It also uses information gathered in the first step to add as many constraints as possible on the variable (length, content, ...) in order to reduce the search space.

The last step, using information from previous stages, generates the Prolog script that will be used by the sequence matcher.

Once the script is generated, the analyzer tries to split the input sequence in smaller parts. Indeed, if grammar analysis or input parameters show match length to be smaller than a fixed integer N, then sequences can be cut in several parts (according to configuration, but at least 2N long). This is used to parallelize the search (multi-threading or using a DRMAA compliant cluster). The analyzer triggers sequence matcher runs on each sequence part and merges the results. In case of multiple input sequences, each sequence analysis will also be parallelized.

If multi-thread is used, the program will limit the number of parallel analysis according to the configuration. If DRMAA is used, the program will also try to use multi-thread on the remote node if sequences can be cut in smaller parts.

• **Sequence Matcher:** The sequence matcher is a Prolog compiled script that loads the script generated by the grammar analyzer. It has been tested with Sicstus Prolog (`sicstus.sics.se`) and SWI-Prolog (`swi-prolog.org`). It scans the input sequence with the input script rules, trying to match each rule one by one. When a complete rule is matched, it records the match.

For each rule element, the matcher takes a chunk of the sequence and tries to apply the rule on the chunk. If it matches, it goes to the end location of the match and tries to apply the next rule. The matcher records all the details of the

match in an XML format. The matcher will optionally apply a filter to delete redundant matches at different levels since it is possible to get two matches at the same location that only differ by their parsing structures.

In case of spacers in a model, the matcher calls an external program using indexing sequence techniques to directly look for positions of the following words. Two possibilities are offered by Logol to perform indexing: VMatch or Cassiopee.

VMatch[1] is a suffix array search tool that supports substitution and indel search. An index is created at startup and the matcher calls the VMatch program to search for a pattern in sequences. VMatch is not open source, but is free for academics. The tool is not delivered with the Logol software suite and needs a manual installation. It is efficient for large sequences.

Cassiopee (`https://github.com/osallou/cassiopee`) is a Ruby tool, developed for *Logol*, though it can be used independently. It scans the sequence to match a pattern with error support. This tool has been developed to provide a complete open source solution, but it is not as efficient as VMatch for large sequences. Then, VMatch usage is the recommended choice when performance is crucial. The tool selection is made in the configuration file so that it can be adapted for each analysis.

4 Illustration: Modeling -1 Ribosomal Frameshifting

RNA recoding is a fundamental biological mechanism that cells use to expand the number of proteins assembled from a single DNA code. There are several types of RNA editing modifying the standard translation of a messenger RNA by the ribosome, which seem largely directed by the 3D conformation adopted by the RNA molecule. The *-1 programmed ribosomal frameshifting* (PRF) is a recoding event which occurs when the ribosome is moving rearward exactly 1 nt on a 'slippery site', `X XXY YYZ`, where `X`,`Y` and `Z` are nucleotides. The ribosome reads the first `X` nucleotide two times in this case. Indeed, while standard translation processes codons `ABX XXY YYZ CDE ...`, the codons processed by PRF are `ABX XXX YYY ZCD ...`. The typical structure promoting a -1 frameshifting event, which we call in the following "PRF pattern", is sketched in Figure 1. It is made sequentially of a start codon, a number of codons, a heptameric slippery site `XXXYYYZ` placed in -1 phase, a few nucleotide spacer, and a characteristic stable secondary structure. The secondary structure is the obstacle stopping the ribosome during the heptamer translation and triggering a movement one nucleotide backwards that causes reading frameshifting. The secondary structure is usually made of a "H-type pseudo-knot" including two nested stem-loops.

A number of tools exists for the detection of putative sites where a -1 frameshifting event might occur [7], but this detection process remains an active research topic since the PRF pattern is not universal (the characteristic features of the heptamer, the spacer, and the secondary structure depend on the organism) and the detection of pseudo-knots is a difficult issue. Many methods proceed by successive filtering steps like KnotInFrame[21], one of the most advanced tool in this category. KnotInFrame initially detects all heptamers `XXXYYYZ`, then

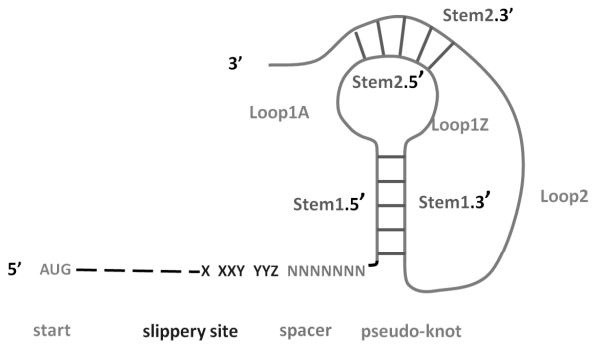

Fig. 1. "PRF pattern"= typical structure promoting a -1 frameshifting event

looks for potential pseudo-knots downstream of this motif, using a dedicated RNA folding procedure.

4.1 PRF Logol Model

The complexity of the PRF pattern makes it a good candidate to investigate the expressivity of Logol. In order to elaborate the corresponding model, one needs to use a number of Logol features like multi-analysis, negative content constraints, repeated motifs or search for biological palindromic structures. We present here the most prominent aspects of the model.

• **Multi-analysis of Two Overlapping ORFs:** Among the mandatory structural features for the occurrence of a -1 frameshifting event, some are concerning the reading frame setting. The standard translation occurs in a sequence with an open reading frame: a start codon (`AUG`) followed by a number of non stop codons (triplets), and a stop codon (`UGA`,`UAG` ou `UAA`) terminating the translation. All these codons are *in 0 phase*. The alternative translation, in case of -1 frameshifting, starts on the same start codon but moves backward one nucleotide on the slippery site, leading to proceed further on triplets *in -1 phase* until a stop codon is reached, also in -1 phase.

In order to possibly generate a -1 frameshifting event, a RNA sequence should thus contain both an open reading frame in 0 phase (a start followed by a sufficient numbers of codons ending by a stop codon) and at a constrained distance from the start, a series of codons ending by a stop in -1 phase.

This check is triggered in Logol by a *multiple analysis* recognizing alternative patterns, "ORF" and "ORFminus", on the same string with a *parameter-passing between the two models* in order to share the common start. The ORF model thus contains a *repeat* that accepts up to 300 non stop codons (a value consistent with the literature), that is: `repeat(notstop(),[0,0])+[0,300]`, where `notstop` stands for a model built from a *view*, that accepts a string of length 3 that is not a `stop`. This is achieved by a *negative content constraint* on the view.

• **Slippery Site and Spacer:** The PRF pattern describes 3 segments: the slippery site, the spacer and the secondary structure. The Logol model for the *slippery site* respects the consensus: it is an heptameric motif in the form XXXYYYZ, which must be positioned in -1 phase, where X is any nucleotide repeated 3 times, Y is the base A or U repeated 3 times, and Z differs from base G, according to the Logol pattern `mod3` below.

```
mod3()==> mod4(), (("aaa")|("uuu")), ! "g":{#[1,1]}
mod4()==> (("aaa")|("ccc")|("uuu")|("ggg"))
```

The *spacer* is straightforwardly described by a gap element (of size less than 10 in this case).

• **Pseudo-knots:** The most efficient secondary structure for -1 frameshifts is the pseudo-knot of type H (two interwined stem-loops, cf 1), even if it is not the sole existing structure. It is thus the structure that has been modeled here.

We first provide a simplified Logol grammar for pseudo-knot structures. In this grammar, `STEM15` refers to the first strand (in the 5' direction) of the first stem and `-"wc" ?IS15` refers to its 2^{nd} strand (in the 3' direction), which is its reverse complement up to 4 mismatches. `STEM25` and `-"wc" ?IS25` refer to the two elements of the2^{nd} stem. The gaps refer to the loop elements between stems.

```
STEM15:{#[4,16],_IS15},.*:{#[1,5]}, STEM25:{#[3,8],_IS25},.*:{#[0,4]},
  -"wc" ?IS15 :{$[0,4]},.*:{#[4,40]},-"wc" ?IS25 :{$[0,2]}
```

Our validation process on real data (next paragraph) resulted in a significant refinement of this model (cf Figure 2). The final model makes use of a great variety of Logol language elements. Among new elements, the model integrates the count of the GC ratio in stems, the separate treatment of nucleotides at the end of stems in order to forbid mismatches at these positions, or the possibility of non-canonical Wobble pairing (G-U), called `wcw` here, at particular stem positions [17]. An excerpt from the Logol grammar dedicated to the first stem follows, the whole model being presented in figure 2:

```
// 50% of GC pairing in Stem1 => 25% of C in [Stem1.5' + Stem1.3']
controls:{ % "c"[mod5.IA5,mod5.IS15,mod5.IZ5,mod5.IZ3,mod5.S13,mod5.IA3]>=25}
mod5()==> (A5:{#[1,1],_IA5},S15:{#[2,14],_IS15},Z5:{#[1,1],_IZ5}):{%"gc":50},
     LOOP1:{#[1,5]},          ... stuff deleted ....
 -"wcw" ?IZ5:{_IZ3}, -"wc" ?IS15:{_S13}:{p$[0,34]}, -"wcw" ?IA5:{_IA3}, ...
```

• **A First Model Validation:** In order to test and refine our -1 frameshift model, we have elaborated a sequence test set [17] around sequences known to produce -1 frameshift events. Thirty proven sequences ("validated -1 frameshift") from the reference base Recode2 (`recode.genetics.utah.edu`) have been completed by random sequences obtained by shuffling 100 times each reference sequence, using Shuffleseq (`emboss.bioinformatics.nl/cgi-bin/emboss/`). This procedure keeps the sequence lengths and nucleotidic ratios of the reference. The final set thus contains 30 "positive" sequences (in which one expects to find the

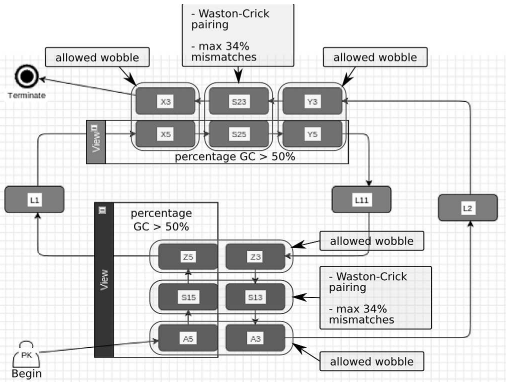

Fig. 2. Global overview of the final Logol model for the PRF pseudo-knot

PRF pattern at the right position) and 300 "negative" sequences (in which one expects a minimum number of PRF pattern occurrences).

This validation work [17] led us to perform comparisons between the Logol prediction of pseudo-knots and those made by "DotKnot", a pseudo-knot prediction software using RNAfold to compute the probability of each folding structure (`http://dotknot.csse.uwa.edu.au`). It brought about some significant changes in our initial model: parameter tuning has concerned wooble pairing, GC ratio and the mismatch rate in the stems. Ultimately, the Logol model finds about 100 matches per sequence on the Recode2 reference set, among them being in most cases the desired match. It is possible to compute a posteriori a quality score for each stem and sort accordingly the matches[17]. This procedure leads to the right frameshift area prediction in 20 cases over the 30. The score is based on a pairing cost function proposed by J.P. Forest [8] for the stem: `{GCpairing=+3, AUpairing=+2, GUpairing=+1, mismatch=-2}`.

• **Runtime.** To give an idea of the performances, parsing the largest reference sequence (30Kb) with the final Logol model takes 1'30s (on a PC Intel X5550, 144Go RAM), while the KnotinFrame answer is immediate on such a sequence. The complete analysis of the Bacillus subtilis genomic sequence (*str168NC_000964.3*, 4.2 Mbp) produces 7000 matches in 2 hours. KnotinFrame web site does not accept such a large sequence.

5 Conclusion

The Logol pattern matching tool has been conceived to allow the modeling and search of realistic structures in biological sequences. It has been designed to be expressive but also evolutive, in order to ease the introduction of new features. The fact that the language has proved fairly well suited to model the complex pattern of ribosomal frameshift, whereas it was not designed for this task, seems an encouraging sign on the genericity of the language elements. The

tool is operational and available on the GenOuest bioinformatics platform, under CeCILL license. Although efforts have been made to offer a wide access to Logol functionalities through either command-line or graphical interface inputs, we welcome any user feed-back to increase its ergonomic features and actual range of applicability.

References

1. Abouelhoda, M.I., Kurtz, S., Ohlebusch, E.: Replacing suffix trees with enhanced suffix arrays. J. Discrete Algorithms 2(1), 53–86 (2004)
2. Billoud, B., Kontic, M., Viari, A.: Palingol: a declarative programming language to describe nucleic acids' secondary structures and to scan sequence database. Nucleic Acids Res. 24(8) (1996)
3. de Castro, E., Sigrist, C.J.A., et al.: Scanprosite: detection of prosite signature matches and prorule-associated functional and structural residues in proteins. Nucleic Acids Research 34(suppl. 2), 362–365 (2006)
4. Dong, S., Searls, D.B.: Gene structure prediction by linguistic methods. Genomics 23(3), 540–551 (1994)
5. Dsouza, M., Larsen, N., Overbeek, R.: Searching for patterns in genomic data. Trends in Genetics 13(12), 497–498 (1997)
6. Eddy, S.: Rnabob: a program to search for rna secondary structure motifs in sequence databases (1996)
7. Firth, A.E., Bekaert, M., Baranov, P.V.: Computational resources for studying recoding. In: Atkins, J.F., Gesteland, R.F. (eds.) Recoding: Expansion of Decoding Rules Enriches Gene Expression, Nucleic Acids and Molecular Biology, vol. 24, pp. 435–461. Springer, New York (2010)
8. Forest, J.P.: Modélisation et détection automatique de sites de décalage de cadre en -1 dans les génomes eucaryotes. Ph.D. thesis, Université de Paris VI (2005)
9. Gattiker, A., Gasteiger, E., Bairoch, A.: Scanprosite: a reference implementation of a prosite scanning tool. Applied Bioinformatics 1(2), 107–108 (2002)
10. Graf, S., Strothmann, D., Kurtz, S., Steger, G.: HyPaLib: a Database of RNAs and RNA Structural Elements defined by Hybrid Patterns. Nucleic Acids Res. 29(1), 196–198 (2001)
11. Jensen, K., Stephanopoulos, G., Rigoutsos, I.: Biogrep: A multi-threaded pattern matcher for large pattern sets (2002)
12. Joshi, A.K., Vijay-Shanker, K., Weir, D.: The convergence of midly context-sensitive grammars. In: Shieber, S.M., Wasow, T. (eds.) The Processing of Natural Language Structure, pp. 31–81. MIT Press, Bosto (1991)
13. Macke, T.J., Ecker, D.J., Gutell, R.R., Gautheret, D., Case, D.A., Sampath, R.: Rnamotif, an rna secondary structure definition and search algorithm. Nucleic Acids Research 29(22), 4724–4735 (2001)
14. Meyer, F., Kurtz, S., et al.: Structator: fast index-based search for rna sequence-structure patterns. BMC Bioinformatics 12(1), 214 (2011)
15. Nicolas, J., Durand, P., et al.: Suffix-tree analyser (stan): looking for nucleotidic and peptidic patterns in chromosomes. Bioinformatics 21(24), 4408–4410 (2005)
16. Pesole, G., Liuni, S., DSouza, M.: Patsearch: a pattern matcher software that finds functional elements in nucleotide and protein sequences and assesses their statistical significance. Bioinformatics 16(5), 439–450 (2000)

17. Rocheteau, A., Belleannée, C.: Recherche d'éléments structurés dans les génomes par modèles logiques. Rapport de recherche PI-1994, Dyliss - Inria - Irisa (April 2012), http://hal.inria.fr/hal-00684388
18. Searls, D.B.: String variable grammar: A logic grammar formalism for the biological language of DNA. Journal of Logic Programming 24(1&2), 73–102 (1995)
19. Searls, D.B., Dong, S.: A syntactic pattern recognition system for DNA sequences. In: Cantor, C.R., Lim, H.A., Fickett, J., Robbins, R.J. (eds.) Proceedings 2nd International Conference on Bioinformatics, Supercomputing, and Complex Genome Analysis, pp. 89–101. World Scientific, Singapore (1993)
20. Strothmann, D., Gräf, S.A., Kurtz, S., Steger, G.: The syntax and semantics of a language for describing complex patterns in biological sequences. Tech. rep., Universität Bielefeld, Arbeitsgruppe Praktische Informatik (August 2000)
21. Theis, C., Reeder, J., Giegerich, R.: Knotinframe: prediction of -1 ribosomal frameshift events. Nucleic Acids Research 36(18), 6013–6020 (2008)