Reasoning about qualitative temporal information with S-words and S-languages

Irène A. Durand
LaBRI, Université de Bordeaux,
idurand@labri.fr

Sylviane R. Schwer[1]
LIPN, Université Paris-Nord
schwer@lipn.univ-paris13.fr

Abstract: Reasoning about incomplete qualitative temporal information is an essential topic in many Artificial Intelligence applications. In the domain of natural language processing for instance, the temporal analysis of a text yields a set of temporal relations between events in a given linguistic theory. Our aim is first to situate the events with respect to each other and to describe (compute or count) all possible relations between them. We first present the formalism of S-languages which formally describes this domain. We explain why Lisp is adequate to implement this theory. Next we describe a Common Lisp system SLS (for S-LanguageS) which implements part of this formalism. A graphical interface written using McCLIM, the free implementation of the CLIM specification frees the potential user of any Lisp knowledge. A complete example illustrates both the theory and the implementation.

1 Introduction

The notion of time is ubiquitous in any activity that requires intelligence. In particular, several important notions like change, causality, and action are described in terms of time. Time has been recognized as a fundamental notion in modeling and reasoning about changing domains. Reasoning about temporal constraints is thus an important task in many areas of computer science and elsewhere, including in scheduling, natural language processing, planning, database theory, diagnosis, circuit design, archeology, genetics, and behavioral psychology [DGV05].

Many frameworks for formalizing time have been proposed, all based on work by logician philosophers who were concerned with physics or language theories, among whom Frege, Prior, Montague, Hamblin, Reichenbach, or Russell, Whitehead and Nicod. This explains why all works have been handled in a logical framework.

In this article, we are concerned with the qualitative aspect of temporal reasoning, *i.e.* only how ”objects” are time-related to each other, without information about any quantitative aspect. We are thus interested in two problems:
(i) a representation problem: how to represent time or temporal objects and what temporal relations are to be represented and how,

[1] Supported by the project ”ANR Blanc Conique”.

(ii) a calculus problem for the reasoning: knowing that a and b are in relation r_1 and b and c are in relation r_2, what possible relations are derived for a and c.

According to the classical spatial representation of time, on a geometrical oriented line, temporal items are taken as points, intervals or chains or points/intervals, depending on whether objects to be represented are viewed as event-like, lasting or iterative. For each representation type, an algebra has been proposed: the point algebra [vBC90] for expressing the three basic relations between points on a line, the point-interval algebra [Vil82] for expressing the five basic relations between a point and an interval on a line, the interval algebra [Ham69] for the thirteen basic relations between intervals on a line, which has become well known since the appearance of [All83]. Other suggested calculi have been derived from one or several of the ones cited above. These algebra are integrated with various logics. There are three known ways of representing and reasoning about temporal information: first order logic, modal logics, and temporal relational calculi. All these approaches are restricted to binary relations and based on transitivity tables like the one for point-point algebra shown in the next table which is read in the following way: the first column shows one of the basic relation between two points p_1 and p_2 on an oriented line: precedes ($<$), equals ($=$) and succeeds ($>$), the first line shows the same for two points p_2 and p_3, and an inside cell provides the possible derived relations between p_1 and p_3. For instance, if $p_1 < p_2$ and $p_2 > p_3$, we can't derive any constraint between p_1 and p_3. But if $p_1 < p_2$ and $p_2 < p_3$, then necessary, we have $p_1 < p_3$.

	$p_2 < p_3$	$p_2 = p_3$	$p_2 > p_3$
$p_1 < p_2$	$p_1 < p_3$	$p_1 < p_3$	$p_1 < p_3$ $p_1 = p_3$ $p_1 > p_3$
$p_1 = p_2$	$p_1 < p_3$	$p_1 = p_3$	$p_1 > p_3$
$p_1 > p_2$	$p_1 < p_3$ $p_1 = p_3$ $p_1 > p_3$	$p_1 > p_3$	$p_1 > p_3$

A qualitative temporal constraint in this framework is depicted in terms of a graph, whose vertices are labeled with temporal objects, and arcs with temporal relations. The consistency of such a graph depends on the calculus with transitivity tables and there is no way to directly express a n-ary relation between n objects.

The use of graphs entailed the resolution of path-consistency and particular complexity problems which gave rise to the exhibition of some subsets of relations (convex, pointizable, Ord-Horn, *i.a.* [NB95]).

The S-languages formalism is based on a totally different approach. It was first introduced in [Sch02]. Its aim was to propose lighter and more intuitive representation than the one given in [Lig91] itself an extension of [Vil82].

Following the natural philosophy of Whitehead, Nicod and Russell, which traces

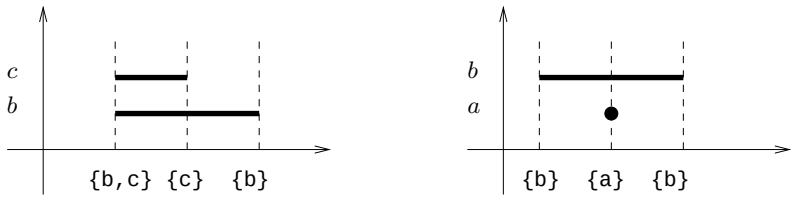

Figure 1: Representation of S-words w_1 and w_2

back to Leibniz[2], a letter a is associated with each temporal object (event, or fact or state) a and acts as its identity. Any object related to a determined scale of time or a point of view can be described as event-like, lasting or repetitive. Let us denote the way the object is to be perceived as its temporal *aspect*. The aim of S-language is to provide a uniform framework for describing both the temporal aspect of objects and their temporal relationships.

If a is depicted as event-like, only one occurrence of its identity will be used; if a is depicted as lasting, two occurrences of its identity will be needed, each of them representing one bound of its interval of duration. If it is depicted as lasting and iterating n times, it will be represented by $2n$ occurrences of its identity.

In a word (a sequence of letters), the fact that a letter b is after another letter a expresses *precedence* between a and b. To express simultaneity, we define *S-letters* (S for Set languages in general or for Synchronization in the framework of time). which are sets of letters occurring at the same time. *S-words* are words over S-letters. Each occurrence of the identity of an object will appear at most once in an S-letter, which is assumed to model a moment.

The S-word $w_1 =$ [{b,c}{c}{b}] contains several pieces of information: there are two temporal objects b and c; they are both lasting objects; b and c start at the same time and b finishes after c. The S-word $w_2 =$ [{b}{a}{b}] means that the temporal object a is event-like and occurs during the lasting object b. A temporal representation of w_1 and w_2 is given in Figure 1.

Given a set of temporal objects, S-words can express constraints over them. Our work focuses on the problem of expressing, enumerating or counting possible scenarii given a set of temporal constraints.

To simplify the presentation of this work, we shall restrict ourselves to handle **lasting** objects – that is lasting and repetitive lasting objects – although taking into account event-like objects does not induce major difficulties.

[2] These philosophers asserted that time is built from nature and that a moment is a "passage of the nature" [Whi20], that is, the set of all events occurring simultaneously at that moment.

2 Preliminaries

2.1 Letters and S-letters

Each temporal object e is represented by a *letter* `e`. By α, we denote the *alphabet* of all letters. It is supposed to be linearly ordered according to the order of the letters in their enumeration. For instance, $\alpha = \{$`a, b, c`$\}$ and `a` $<$ `b` $<$ `c`. $\#\alpha$ denotes the cardinality of α.

An *S-letter* is a non-empty subset of α. It defines synchronization points between events. For instance, `{a,c}` is an S-letter meaning that `a` and `c` occur simultaneously. By $S_\alpha = \mathcal{P}(\alpha) \setminus \{$`{}`$\}$, we denote the set of S-letters whose *underlying* alphabet is α and name it the S-alphabet of all letters. For instance,

$S_{\{\mathtt{a},\mathtt{b},\mathtt{c}\}} = \{$`{a}, {b}, {c}, {a,b}, {a,c}, {b,c}, {a,b,c}`$\}$.

2.2 S-words

An *S-word* is a sequence of the S-letters, in other words an element of $(S_\alpha)^*$. We surround the sequences of the S-letters of an S-word by brackets (`[ ]`). The *Parikh vector* of a S-word w is the vector $\overrightarrow{w}$ of $\mathbb{N}^{\#\alpha}$ whose i^{th} coordinate is the number of occurrences of the i^{th} letter. The alphabet $\alpha(w)$ of a S-word w is the set of letters appearing in its S-letters.

Example 1. let $\alpha = \{$`a, b, c, d`$\}$ and $w =$ `[{a}{a,b}{a,c}{a,b,c}]`.
$\alpha(w) = \{$`a, b, c`$\}$ and $\overrightarrow{w} = (4, 2, 2, 0)$.

Letters in S-letters of an S-word w can be *marked* with the following bijective marking. For all letters $l \in \alpha$ appearing in w, the first occurrence of l is marked 0, the second 1 and so on. Marking the S-word w gives:

`[{a_0}{a_1,b_0}{a_2,c_0}{a_3,b_1,c_1}]`.

A possible meaning for S-words is the following. The marked letter l_0 (and each other appearance of l with an even mark) indicates that the object associated with the letter l starts. Each l_i with an odd mark indicates that the object stops.

As the marking of an S-word is bijective, we generally don't write marks. However, when dealing with subwords of S-words — which happens when handling incomplete temporal descriptions — it will be informative to write the marks. For instance, given two lasting objects a and b, "b starts strictly after the end of a" could be written by the complete S-word `[{a}{a}{b}{b}]`
(which is implicitly `[{a_0}{a_1}{b_0}{b_1}]`)
or only described by `[{a_1}{b_0}]` as `[{a_0}{a_1}]` and `[{b_0}{b_1}]` are implicit.

Hence our letters in S-letters are always marked (either implicitly or explicitly). The *marked-alphabet* $\alpha_M(L)$ of an S-word w is the set of marked (or implicitly marked) letters of w.

2.3 S-languages

An *S-language* is a set of S-words. Given an alphabet α, and a vector $\vec{w} \in \mathbb{N}^{\#\alpha}$ (which associates an integer to each letter), the set of all possible S-words is called the *S-universe* of α and $\vec{w}$ and is denoted by $\mathcal{U}(\alpha, \vec{w})$. Given an S-language L, the alphabet $\alpha(L)$ of L is the set of letters of L and the marked-alphabet $\alpha_M(L)$ of L is the set of marked-letters appearing in the S-words of L.

If, by hypothesis, we know that each object associated with a letter l occurs a finite number of times n then for each letter `l`, we have a maximum index of $2n-1$. In that case, S-words have *finite* length and the S-universe of interest is the one associated with the vector having as its i^{th} coordinate the length of the S-word that depicts it. In these cases, we do not mention the Parikh vector of the S-universe.

But S-languages are not always restricted to a finite S-universe: in [Sch07a] S-languages over infinite S-words are used to deal with execution traces in distributed systems. In this case the alphabet is finite but the alphabet of marked-letters is not (the set of possible marks is infinite) and the Parikh vector cannot be defined, but the S-universe is defined as the set of all possible S-words and is infinite.

An S-language will be represented either in *extension*, *i.e.* by giving the list of its S-words (this is possible in the finite case only: finite language of finite S-words) or by expressions over S-languages, which are called *S-expressions* (not to be confused with `Lisp` Sexpressions) using operators. Operations and expressions over S-languages will be presented in Section 3.

Suppose for instance that we have two independent objects a and b, each occurring once. They are represented by S-words `[{a}{a}]` and `[{b}{b}]` respectively. The S-universe is the S-language containing all the possibilities of combining these two objects that is all the S-words having $(2, 2)$ as Parikh vector. This S-language has 13 S-words, given by the Delannoy number $D(2, 2)$ [Slo], which depicts the 13 possible relationships between two intervals on a line and well-known in artificial reasoning community as Allen's relations [All81, All83]. We shall see in Section 3 that this S-language can be represented by the *mix* of the two S-words: `[{a}{a}]` X `[{b}{b}]`.

So, for just *two* objects a and b, each occurring once and without any specific constraint (other that "the beginning of an object occurs strictly before its end"), we have a set of 13 possibilities (S-words) for combining them. Now if constraints exist between the objects, we will get an S-language which is a subset of these 13 possibilities. Each subset of the S-universe corresponds to specific constraints.

Example 2. For instance, if we add the constraints that b must start strictly after a and end after than or at the same time as a, we get the following S-language with 4 possibilities: $L_1 =$ `{[{a}{b}{a,b}], [{a}{a}{b}{b}], [{a}{a,b}{b}], [{a}{b}{a}{b}]}`.

There are 2^{13} S-languages included in the S-universe; the whole part represents the absence of constraint; the empty part represents incompatible constraints.

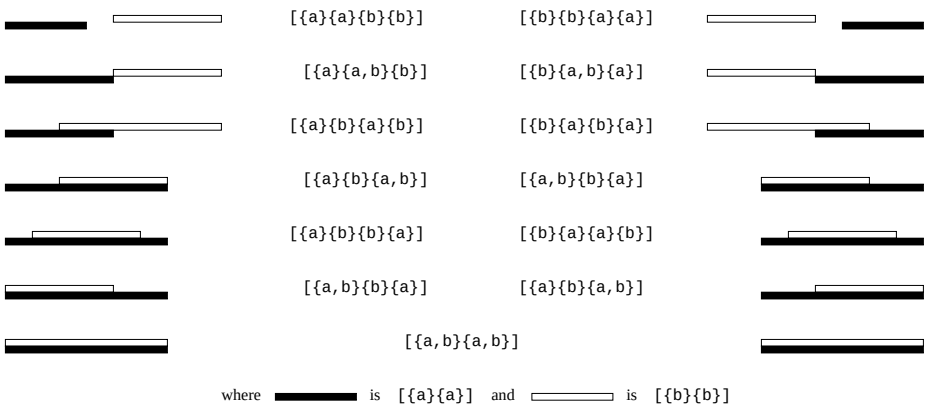

Figure 2: The 13 relations between two intervals on a line

3 Operations and expressions on S-languages

3.1 Classical operations on languages

From one point of view, S-languages are a special case of formal languages. Consequently, all classical operations on formal languages apply [RS96]. In particular, the boolean operations (union, intersection, complement), concatenation, mirror are defined in the usual way considering that S-letters are the letters of the S-words. In the classical framework, letters are basic objects which cannot be decomposed. In the S-languages framework, the letters of the S-words are S-letters, *i.e.* sets of letters which we may want to compose or decompose. Expressions over S-languages will be refered to as S-expressions (not to be confused with `Lisp` Sexpressions (`Sexpr`. The classical projection would be to project over a sub-alphabet of S-letters: it erases S-letters.

3.2 S-projection

In the S-language framework, we may define the S-projection over a sub-alphabet of letters which erases letters inside the S-letters of a S-word. The same extension can be considered for morphisms and inverse morphisms.

The *S-projection* of an S-word w over the alphabet α, denoted by $w_{|\alpha}$ is the S-word obtained by erasing from w all occurrences of letters which are not in α and then every S-letter which has become empty.

Example 3. Let $w =$ `[{a,c}{a,b}{c,d}{a,b,c}]` and $\alpha = \{$`a`, `b`$\}$.
$w_{|\alpha} =$ `[{a}{a,b}{a,b}]`.

The S-projection of an S-language is the set of the S-projections of its S-words.

3.3 The join operation

Consider two S-languages L_1, L_2 over respective alphabets $\alpha(L_1)$ and $\alpha(L_2)$. Each S-language L_i represents temporal constraints which restrict the S-universe $\mathcal{U}(\alpha(L_i))$. *joining* the two languages L_1 and L_2 consists in constraining $\mathcal{U}(\alpha(L_1) \cup \alpha(L2))$ with the union of the constraints of both languages. The join operation will be denoted by the symbol `J`.

Example 4. Recall $L_1 =$ `{[{a}{b}{a,b}], [{a}{a}{b}{b}], [{a}{a,b}{b}], [{a}{b}{a}{b}]}` of Example 2. The language $L_2 =$`{[{a}{a,c}{c}]}` can be described by the constraint "c starts when a stops". L_1`J`L_2 yields the S-language
`{[{a}{b}{a,c}{b,c}], [{a}{b}{a,c}{b}{c}], [{a}{b}{a,c}{c}{b}],`
`[{a}{b}{a,b,c}{c}], [{a}{b}{b}{a,c}{c}]}`.

There are two special cases for the join operation: the first case occurs when the alphabets of the two languages are identical, then the join corresponds to the intersection of the two languages; the second case occurs when the alphabets are disjoint and is described below.

3.3.1 The mix operation (join with disjoint alphabets)

In the case of disjoint alphabets, the join operation is a kind of *shuffle* that we call *mix* and denote by `X`: it considers all possibilities of ordering independent letters.

In the case where $L_1 =$ `[{a}{a}]` and $L_2 =$ `[{b}{b}]`, the S-language corresponding to L_1`J`L_2 (already seen in Section2.3) can be obtained by applying the two rewrite rules

```
{a}{b} -> {a,b}
{a,b} -> {b}{a}
```

on the concatenation of L_1 and L_2 which is `[{a}{a}{b}{b}]`. The lattice (shown in Figure 3.3.1) obtained by applying the rewrite rules contains all the S-words of L_1 `X` L_2. Note that these S-words are the same as the one in Figure 2.3.

This principle generalizes to any number of letters and S-languages with any cardinality.

A mix expression is a compact way of representing an S-universe (all the possibilities for a given set of temporal objects). S-universes are usually very big (so big that we can't compute them in practice) so the mix is an indispensable tool to handle S-languages. Very often the computation of the language corresponding to a mix expression will lead to a combinatorial explosion. Consequently, such computation should be avoided as much and as long as possible. The idea is to first perform every possible simplifications which could prune part of the search space.

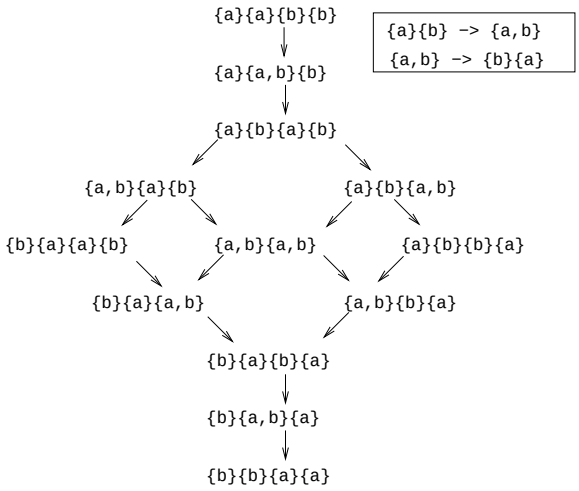

Figure 3: Lattice of the mix operation

3.3.2 Join operation (with intersecting alphabets)

In the general case, the alphabets have a non-empty intersection ($\alpha(L_1) \cap \alpha(L_2) \neq \emptyset$).

The basic operation is defined on S-words. Let f and g two S-words.

If $\alpha(f) \cap \alpha(g) = \emptyset$ then $f \,\mathtt{J}\, g = f \,\mathtt{X}\, g$ as defined above. Otherwise, let $\beta = \alpha(f) \cap \alpha(g) \neq \emptyset$. If the projections $f_{|\beta}$ and $g_{|\beta}$ differ then the constraints inherent to the two words are incompatible and $f \,\mathtt{J}\, g = \{\}$. Otherwise, the S-words are compatible and $f \,\mathtt{J}\, g$ is the S-language containing all S-words h written over $\alpha(f) \cup \alpha(g)$ which satisfy $h_{|\alpha(f)} = f$ and $h_{|\alpha(g)} = g$. Let for instance
$f =$ `[{a,c}{a,b}{c,d}{a,b,c}]` and
$g =$ `[{e}{a,e,f}{e}{a,b}{f}{a,b,f}{e}]`.
Then $\beta =$ `{a,b}`, $f_{|\beta} =$ `[{a}{a,b}{a,b}]` $= g_{|\beta}$ and

```
f J g = {[{e}{a,c,e,f}{e}{a,b}{c,d,f}{a,c,b,f}{e}],
         [{e}{a,c,e,f}{e}{a,b}{c,d}{f}{a,c,b,f}{e}],
         [{e}{a,c,e,f}{e}{a,b}{f}{c,d}{a,c,b,f}{e}]}.
```

However, we can give a more compact representation using the mix operation:
`[{e}{a,c,e,f}{e}{a,b}] . ([{c,d}] X [{f}]) . [{a,b,c,f}{e}]`

The join operation extends to languages: the join of two S-languages is the union of the joins of an S-word of the first language and an S-word of the second. A description of the algorithm can be found in [Sch07b]. Our implementation provides both a recursive and an iterative version of it. The join algorithm is a crucial in the S-languages setting because solving a problem described by a set of constraints `{E1, E2, ... En}` consists in computing the S-language corresponding to the S-expression
`E = E1 J E2 J ... J En`.

3.4 Example

The following example is inspired by [Rev96]. Consider a set of 6 trains named $\{A, B, C, D, E, F\}$ with the following set of temporal constraints.

1. A, B and E reach the platform at the same time
2. A leaves before B.
3. A leaves after or at the same time as C but before the arrival of D.
4. D and F arrive at the same time as B is leaving.
5. E and D leave at the same time.

We consider the following problem: how many platforms are necessary to satisfy constraints 1 to 5. We formalize the problem into the S-languages framework. For each train, we consider the event corresponding to the time during which the train remains at the platform. Because of security reasons, we do not allow that a train to arrive on a track from which a train is currently leaving.

Our alphabet is $\alpha = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}, \mathtt{e}, \mathtt{f}\}$, one letter for each train. The S-universe is the S-language represented by the following mix expression

```
[{a}{a}] X [{b}{b}] X [{c}{c}] X [{d}{d}] X [{e}{e}] X [{f}{f}]
```

which means that we have 6 lasting temporal objects. The S-universe contains

$$D(2, 2, 2, 2, 2, 2) = D(2^6) = 308682013$$

S-words [Slo]. The five constraints can be expressed by the following five S-expressions:

```
1. E1 = [{a,b,e}] . ([{a}] X [{b}] X [{e}])
2. E2 = ([{a}] X [{b}]) . [{a}{b}]
3. E3 = (([{a}] X [{c}{c}]) . [{a}{d}{d}]) U
        (([{a}] X [{c}]) . [{a,c}{d}{d}])
4. E4 = [{b}{b,d,f}] . ([{f}] X [{d}])
5. E5 = ([{e}] X [{d}]) . [{d,e}]
```

3.5 Simplifying S-expressions

For solving a set of constraints `{E1, E2, ... En}`, one must evaluate the S-expression `E1 J E2 ... J En`. In general, it is not tractable to evaluate the S-languages `Li` corresponding to the `Ei` and then joining them because the intermediate S-languages are much too big. The key idea is to simplify to `e` until it becomes reasonable to compute the final S-language. Finding simplifications and proving they are correct is a difficult domain which is not completely explored. The first kind of simplifications results from classical properties of the operators like associativity, commutativity, idempotence and distributivity. The other simplifications concern the join operation or its special cases (mix, intersection). For instance, the intersection of two languages with disjoint alphabets is empty; the join of a language with its S-universe is the language itself.

For our trains example of Section 3.4, `SLS` is able to simplify the S-expression which evaluates to an S-language containing 24 S-words of length between 5 and 7. The final language can be written usig mix as:

```
E = ([{c}{c}] X [{a,b,e}]).[{a}].[{b,d,f}].([{f}] X [{e,d}]) U
     ([{a,b,e}] X [{c}]).[{a,c}].[{b,d,f}].([{f}] X [{e,d}])
```

In order to solve our problem, we have to recall the good interpretation of what this S-language depicts (the possible relationships between the periods where trains are stopped at a platform), then to find inside `E` an S-word which minimizes the meeting or interleaving between these periods. The first choice is to take `([{c}{c}{a,b,e}])` from the left sub-S-expression `([{c}{c}] X [{a,b,e}])` which isolates the train `C`. The new S-expression is `E'=[{c}{c}{a,b,e}{a}{b,d,f}].([{f}] X [{e,d}])` and contains only 3 S-words. First, `C` stops and leaves, then `A`, `B`, `E` arrive all at the same time, then we need at least three tracks. But `A` leaves only before the arrival of `D` and `F`, then we need one more track. The answer of the problem is then that 4 tracks are enough; 4 tracks are also sufficient for all S-words of `E'`.

4 Implementation of S-languages

It will not take long to justify the choice of the Common Lisp language to implement the theory of S-languages: the domain is typically symbolic as opposed to numeric; the data are highly hierarchical which justifies an object-oriented language; in addition, multiple inheritance is very useful for factoring properties and associated methods for simplifying S-expressions.

4.1 Implementation of basic objects

The basic `SLS` objects are letters `(letter)`, marked letters `(mletter)`, S-letters `(sletter)`, alphabets for all the different kinds of letters `(alphabet, malphabet)`, S-words `(sword)`.

To prevent combinatorial explosion we use the well-known technique of *hash-consing*: each element of each object category is represented by a unique Lisp object; there is a list for each category of object; the objects are stored in the list corresponding to its category. When the creation of an object is required, a look-up is done in the corresponding list; if an object with equal components (in the `eq` sense) is found such object is returned; otherwise a new object is constructed and stored in the list. Here is the example of the `mletter` case.

```
(defmethod make-mletter ((string string) &optional (mark 0))
  (let* ((letter (make-letter string))
         (name (name letter)))
    (or (find-object name (mletters *spec*)
                     :test (lambda (name mletter)
                             (and (eq name (name mletter))
                                  (= mark (mark mletter)))))
        (let ((mletter (make-instance 'mletter :letter letter
                                      :mark mark)))
          (setf (mletters *spec*)
                (append (mletters *spec*) (list mletter)))
          mletter))))
```

This technique has also the advantage that SLS basic objects are `eq`-comparable which improves time performance.

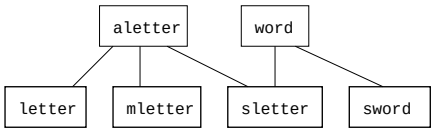

Figure 4: Classes for basic `SLS` objects

The hierarchy of the classes describing basic `SLS` objects is presented in Figure 4.1. Note that an S-letter, being a sequence of letters, is itself a word (but not an S-word).

4.2 Implementation of S-expressions

The class `alanguage` contains all objects which describe languages. A language can be represented by its set of words (`language`, `word`) or by an expression. An expression is defined recursively: it is either a concrete `language` or an expression with an operator and whose arguments are expressions. Note the use of the `mixin`

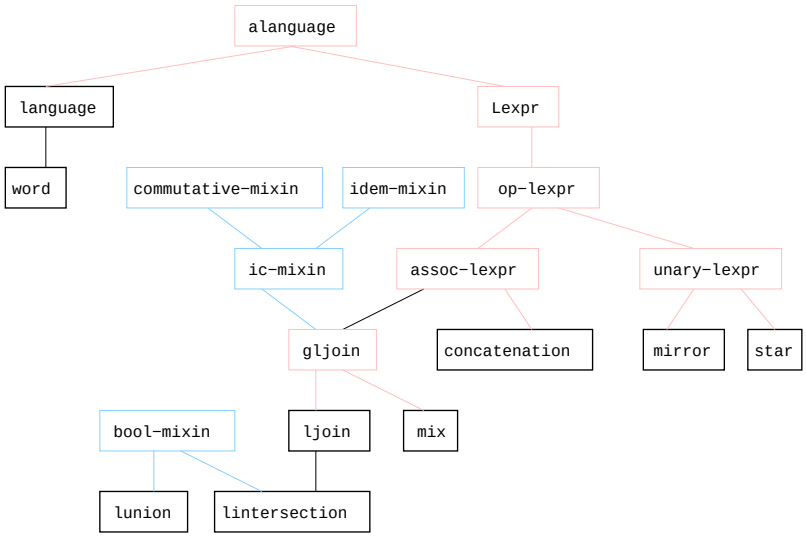

Figure 5: Class hierarchy for representing S-languages

classes to capture properties which help simplifying expressions. For instance, the pri-

mary method `clean-args` normalizes the arguments of an associative S-expression. The secondary methods complete this task according to the other properties of an operator. For instance, if the operator is idempotent, we can remove duplicated or equivalent arguments.

```
(defmethod clean-args ((lexpr assoc-lexpr)) ...)
(defmethod clean-args :before ((lexpr fold-mixin))
  (setf (args lexpr)
        (remove-duplicates (args lexpr) :test #'equivalent))
  lexpr)
```

4.3 Specifications for SLS

SLS handles a set of specifications that can be loaded interactively. A specification consists of a signature, possibly a set of variables, followed by a list of SLS objects. SLS objects are S-words, S-expressions, S-languages, Problems (set of S-expressions which correspond to constraints). In a same specification, one stores objects from a common S-universe.

Figure 4.3 shows an example of such a specification. That specification contains the train problem of Section 3.4. It also shows how to specify S-word, S-expressions or S-languages in extension.

Figure 6: Example of an SLS specification

4.4 The graphical interface

A graphical user interface helps the user load his/her data (S-words, S-expressions, S-languages) and apply operations on it. It is written using the McCLIM[SM02] system which is the free implementation of the CLIM specification. A snapshot of the SLS window after loading the `train.txt` specification is shown Figure 7. All the commands are either accessible from the command line in the top window or from menus, classified according the type of object they operate on. Here we have applied the command `Solve` (also in the `Problem` menu) which transforms the set of constraints of the problem into a (when possible) simplified S-expression which becomes the current S-expression. Next we have applied the `Slanguage Sexpr` command (also in the `Sexpr` menu) which computes the S-language corresponding to the current S-expression and invoked the `Cardinality Slanguage` command (also in the `SLanguage` menu) which prints the cardinality of the current S-language. Finally, with the `Membership To Slanguage`, we verify that the current S-word belongs to the current S-language. The final look of the window is shown in Figure 8.

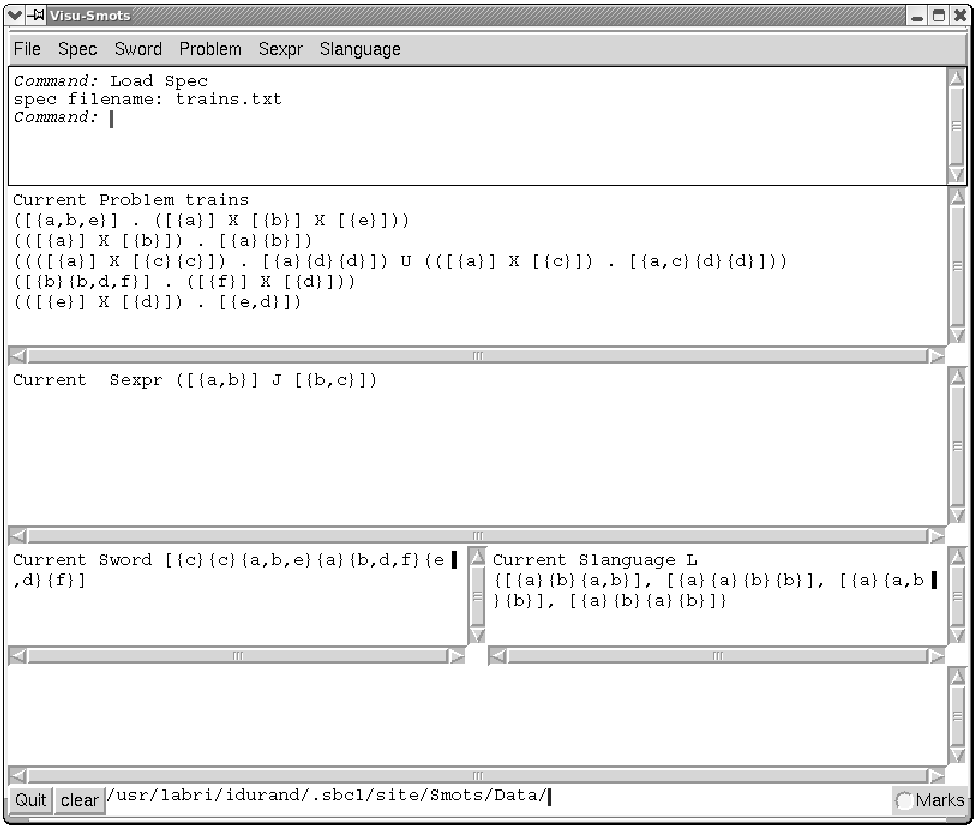

Figure 7: First snapshot of SLS

SLS contains altogether 6000 lines of Common Lisp of which around 1200 correspond to the graphical interface. One the project page, `http://dept-info.labri.u-bordeaux.fr/~idurand/SLS/`, one can find a description of the project, a User's Manual, an archive with the latest source and executable files for a few architectures.

5 Related work and perspectives

Objects and temporal constraints between them is a crucial matter in many domains (artificial intelligence, linguistics, music,...). Making our software really usable in applications work requires work in two directions.

The problem of constraint satisfaction is intrisically exponential. In S-languages, the mix operation is a way to avoid combinatorial explosion in some cases. For the other cases, and in order to minimize the risk of combinatorial explosion, theoretical

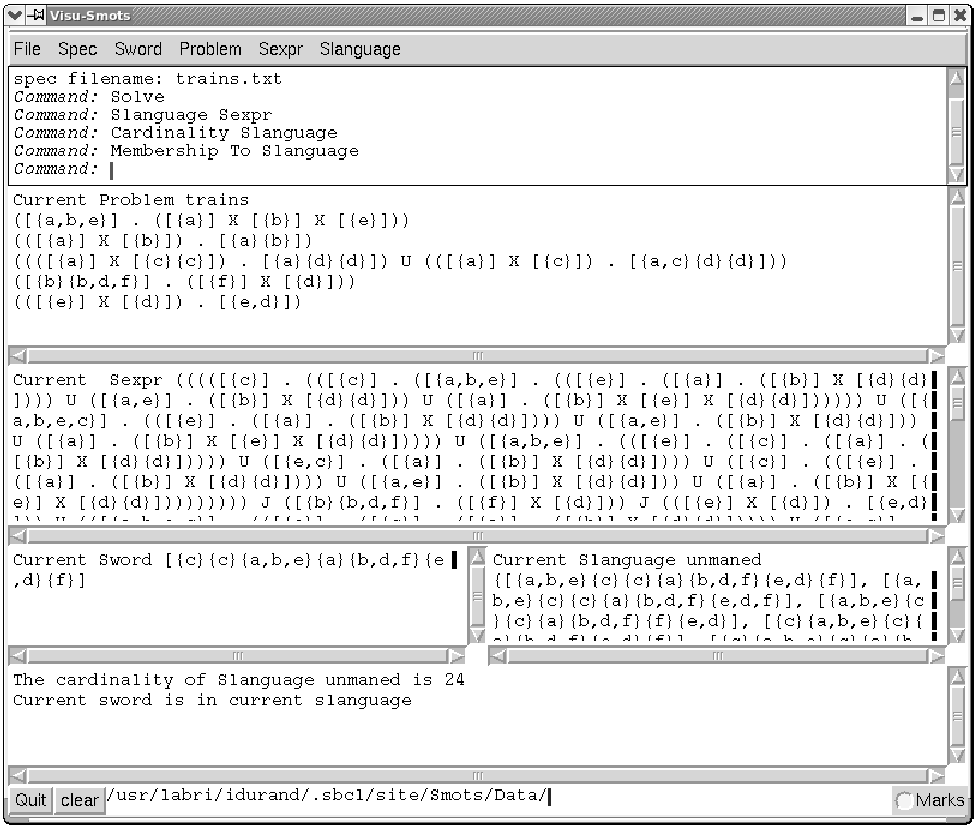

Figure 8: Second snapshot of SLS

work must be done for better simplifying S-expressions before calculating in extension the corresponding S-language. When we can't avoid combinatorial explosion, programming should be as efficient as possible in terms of memory allocation and time computations. Many improvements may be done in that direction, particularly we haven't yet exploited the possibility of detecting and sharing equivalent expressions as we already do for S-word, S-letters. Futhermore, we also plain to analyse in terms of S-expressions, the convex, pointizable and Ord-Horn classes studied in the interval algebra theory [NB95].

At the outside level, much work needs to be done to allow non-computer scientists to use the tool. Representing graphically S-words could be a first step. Next we could think of a tool for helping the user defining graphically constraints between objects resulting in a set of S-words.

Acknowledgements

The authors would like to thank the referees for their constructive reports and Lucas Saiu for his careful rereading.

References

[All81] James F. Allen. An interval-based representation of temporal knowledge. In *Proceedings of the Seventh International Joint Conference on Artificial Intelligence*, pages 221–226, 1981.

[All83] James F. Allen. Maintaining knowledge about temporal intervals. *Commun. ACM*, 26(11):832–843, 1983.

[AS03] Jean-Michel Autebert and Sylviane R. Schwer. On generalized delannoy paths. *Journal on Discrete Mathematics*, 16(2):208–223, 2003.

[DGV05] Mickael D. David, Dov M. Gabbay, and Lluis Vila. *(eds)*. Elsevier, 2005.

[Ham69] C. L. Hamblin. Starting and stopping. *The Monist*, 53(3):410–425, 1969.

[Lig91] Gérard Ligozat. On generalized interval calculi. In *AAAI*, pages 234–240, 1991.

[NB95] Bernhard Nebel and Hans-Jürgen Bürckert. Reasoning about temporal relations: A maximal tractable subclass of allen's interval algebra. *Journal of the ACM*, 42(1):43–66, 1995.

[Rev96] Joel Revault. *Une modélisation par le graphe de la relation meet pour traiter des contraintes temporelles exprimées à l'aide d'intervalles.* Phd thesis, Université de Nantes, 1996.

[RS96] G. Rozenberg and A. Salomaa. *Handbook of Formal Languages: Word, Language, Grammar*, volume 58 of *Lecture Notes in Computer Science*. Springer, 1996.

[Sch02] Sylviane R. Schwer. S-arrangements avec répétition. *Comptes Rendus de l'Académie des Sciences, Mathématiques*, 4:261–266, 2002.

[Sch07a] S. Schwer. Temporal reasoning without transitive tables. arXiv:0706.1290v1 [cs.AI], June 2007.

[Sch07b] Sylviane R. Schwer. Traitement de la temporalité des discours : une analysis situs. In *Information temporelle, procédures et ordre discursif*, volume 18 of *Cahiers Chronos*. Rodopi, Amsterdam, 2007.

[Slo] Neil Sloane, editor. *The On-Line Encyclopedia of Integer Sequences*, chapter A055203. http://www.research.att.com/ njas/sequences/.

[SM02] Robert Strandh and Tim Moore. A free implementation of clim. In *Proceedings of the International Lisp Conference*, San Francisco, California, October 2002.

[vBC90] P. van Beek and R. Cohen. Exact and approximate reasoning about temporal relations. *Computational Intelligence*, 6:132–382, 1990.

[Vil82] Marc Vilain. A system for reasoning about time. In *Proceedings of the AAAI*, pages 197–201, 1982.

[Whi20] Allan North Whitehead. *The concept of nature*. Cambridge University Press, Cambridge, 1920.