D-SPACES: An Implementation of Declarative Semantics for Spatially Structured Information

Stefan Haar, Salim Perchy, Frank Valencia

D-SPACES: An Implementation of Declarative Semantics for Spatially Structured Information*

Stefan HAAR
INRIA, LSV École Normale Supérieure de Cachan
stefan.haar@inria.fr

Salim PERCHY
INRIA, LIX Université Paris-Saclay
yamil-salim.perchy@inria.fr

Frank VALENCIA
CNRS, LIX Université Paris-Saclay & Universidad Javeriana de Cali
frank.valencia@polytechnique.fr

Abstract

We introduce in this paper D-SPACES, an implementation of constraint systems with space and extrusion operators. Constraint systems are algebraic models that allow for a semantic language-like representation of information in systems where the concept of space is a primary structural feature. We give this information mainly an epistemic interpretation and consider various agents *as entities acting upon it. D-SPACES is coded as a* `c++11` *library providing implementations for constraint systems, space functions and extrusion functions. The interfaces to access each implementation are minimal and thoroughly documented. D-SPACES also provides property-checking methods as well as an implementation of a specific type of constraint systems (a boolean algebra). This last implementation serves as an entry point for quick access and proof of concept when using these models. Furthermore, we offer an illustrative example in the form of a small social network where users post their beliefs and utter their opinions.*

1. Introduction

Systems where information is created and manipulated across a spatial structure are now commonplace. Applications like social networks, forums, or any other that organizes its information in a defined hierarchy are among these systems. The nature of this information can be reviews, opinions, news, etc., whereas the information belongs to a certain entity, e.g. users, agents, applications. This relation of ownership and its alteration can be conceptualized as *space* and *extrusion* respectively [1, 2]. We aim to achieve a clear understanding of the concept of space and extrusion that will enable us to study the meaning of information in these systems.

In this work we focus in epistemic systems where we have agents *believing* information [3] and uttering opinions and lies [4]. In order to attain a semantical meaning of these epistemic behaviors we use declarative models, specifically *constraint systems*; algebraic structures that operate on elements called constraints [5] (the information). Constraints can be viewed as assertions representing partial information (e.g. $t < 50$ would stand for a certain temperature being lower than 50 degrees), this makes constraint systems ideal to model and operate over scattered data.

The characterization of space as the operator $[\cdot]$ over constraints was developed in [1]. This made possible assertions like $[c]_i$ "information c belongs to agent i's space" or $[[c]_j]_i$ "c holds in a space associated to agent j inside agent i's space". Alternatively, we can epistemically interpret these assertions as "agent i believes c" and "agent i believes agent j believes c" respectively. Movement of information across spaces was introduced by means of the constraint operator $\uparrow$ called *extrusion* [2]. One can now conceive statements like $[\uparrow_i [c]_j]_i$ "agent i extrudes the information c to agent j"

*This research was partially supported by Labex DigiCosme (project ANR-11-LABEX-0045-DIGICOSME) operated by ANR as part of the program "Investissement d'Avenir" Idex Paris-Saclay (ANR-11-IDEX-0003-02)

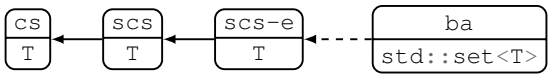

Figure 1. General class diagram

or epistemically as "agent i extrudes that agent j believes c".

The purpose of this article is to present D-SPACES, an implementation of constraint systems with space and extrusion operators. Our intention is to use it to provide a semantic language for describing systems where information is structured in a hierarchy of spaces. D-SPACES was conceived as a `c++11` library, using the boost graph library[1] (`BGL`). It is thoroughly documented using `HeaderDoc`[2] and can be directly used in the OS X (XCode) and Linux (GCC+Make) environments[3]. Usage on Windows depends upon compilation of `BGL`, nonetheless the implementation is sufficiently cross-platform.

The paper is divided in four sections. Section 1 provides an introduction with basic details of the tool. Section 2 defines the constraints systems and describes D-SPACES; it explains the general design, the class interfaces, and details the property checking methods implemented therein. Moreover, we provide some remarks on the complexity issues of the implementation of some constraint system operations. In Section 3 we present powerset boolean algebras as a specific instantiation of our model and we use it to give semantics to a belief language describing a small social networks based on tags. Finally, Section 4 offers some concluding remarks and future endeavors regarding D-SPACES.

2. Implementing Space and Extrusion in Constraint Systems

We begin by describing the class hierarchy of D-SPACES (Figure 1). There are three modules implementing each constraint system, they are named `cs`, `scs` and `scs-e`. A fourth module, named `ba`, implements powerset boolean algebras using the functionality of all the others.

Each module parametrizes the cs elements (the information) using a template `T`. The type used must be comparable in the standard way (i.e. `operator<`) as there is reliance on the automatic sorting of the container `std::set`. Currently, there are instantiations of types `int`, `char`, `std::string` and containers `std::vector` and `std::set` of these same types. We continue with the description of the `cs` module.

[1] `http://www.boost.org/doc/libs/release/libs/graph/`

[2] `http://developer.apple.com/opensource/tools/headerdoc.html`

[3] `http://www.lix.polytechnique.fr/~perchy/d-spaces/`

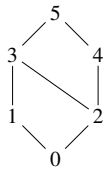

Figure 2. A Poset

Flat Constraint Systems. We first formalize the concept of *constraint system*. A basic background in domain theory is presupposed [6, 7].

Definition 2.1 (Lattice)**.** *A* lattice *is a partially ordered set (poset)* $(Con, \sqsubseteq)$ *where for each* $c, d \in Con$ *we define; (i)* $c \sqcap d$ *(read as the* meet *of* c *and* d*) as the maximal element* e *w.r.t.* $\sqsubseteq$ *s.t.* $e \sqsubseteq c$ *and* $e \sqsubseteq d$ *and, (ii)* $c \sqcup d$ *(read as the* join *of* c *and* d*) as the minimal element* e *w.r.t.* $\sqsubseteq$ *s.t.* $c \sqsubseteq e$ *and* $d \sqsubseteq e$*.*

The ordering relation in posets is *reflexive* (i.e. $c \sqsubseteq c$), *antisymmetric* (i.e. $c \sqsubseteq d$ and $d \sqsubseteq c$ imply $c = d$) and *transitive* (i.e. $a \sqsubseteq b$ and $b \sqsubseteq c$ imply $a \sqsubseteq c$). Its reverse is denoted as $\sqsupseteq$. The meet and join operators are alternatively called *greatest lower bound* (glb) or *infimum* and *least upper bound* (lub) or *supremum*. We give an example lattice with elements $\{0,1,2,3,4,5\}$ where Figure 2 is the Hasse diagram of its underlying poset.

Definition 2.2 (cs)**.** *A* constraint system *[5] is a complete lattice, that is, a lattice where the meet and join operations are defined for every subset of Con.*

Intuitively, a cs is an information structure where its elements are the set *Con* (called *constraints*). A cs has a bottom element *true* (denoted as the global meet $\bot$ in lattice literature) and a top element *false* (denoted as $\top$). Furthermore, the reverse ordering relation $\sqsupseteq$ is interpreted in cs as *entailment* (i.e. $d \sqsupseteq c$ means d entails c). Notice this interpretation suggests the greater an element is on the ordering relation $\sqsubseteq$, the more information the element denotes. In the example of Figure 2, *true* is the element 0 and *false* is the element 5, needlessly to say, *false* entails all the elements of the cs.

We can also define a binary implication operator $c \to d = \bigsqcap \{e \mid c \sqcup e \sqsupseteq d\}$. This definition is adapted from *Complete Heyting Algebras* [8] and it additionally allows us to encode the pseudo-complement of a constraint as $\sim c = c \to false$. Pseudo-complements do not necessarily comply with the law of the excluded middle

Table 1. Interface to `cs`

Method	Desc.	Symbol
`add_element( T c, vector<T> L, vector<T> U )`	addition of element	$L \sqsubseteq c \sqsubseteq U$
`bool leq( T c, T d )`	ordering relation	$c \sqsubseteq d$?
`T glb( vector<T> elems )`	meet of elements	$\bigsqcap(elems)$
`T lub( vector<T> elems )`	join of elements	$\bigsqcup(elems)$
`T imp( T c, T d )`	implication operator	$c \to d$

and the above definition only works as an implication if the cs is distributed, that is if for every $a,b,c \in Con$ we have that:

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c).$$

Interface and usage. Table 1 describes part of the interface to the `cs` module. The input elements in `glb` and `lub` (the respective implementations of the meet and join operators) may be empty vectors, in this situation they produce the bottom and top elements respectively. Similarly, in the method `add_element` the upper and lower bounds of `c` (i.e. parameters `L` and `U`) may be empty, denoting *false* and *true* respectively.

Properties of cs. As mentioned above, one optional and important property of constraint systems is that of *distributivity*. This property is necessary for *modus ponens* to hold, that is, $(c \to d) \sqcup c \sqsupseteq d$ must be true for every $c,d \in Con$ [2]. In D-SPACES distributivity can be checked with the boolean method `CS.is_distributive()`.

The D-SPACES snippet in Figure 3 creates a constraint system with the underlying lattice of Figure 2. Additionally, it calculates $1 \sqcup 2$, $\sqcap\{2,3,4\}$, $2 \to 3$ and checks if the cs is distributive.

Spatial Constraint Systems with Extrusion. We continue with the `scs` and `scs-e` modules. For this we begin by defining the remaining two constraint systems.

Definition 2.3 (scs)**.** *An n-agent* spatial constraint system (scs) *is a cs equipped with n self-maps* $[\cdot]_1, \ldots, [\cdot]_n$ *(called space functions) over its set of elements. Additionally, for each map* $[\cdot]_i : Con \to Con$ *we have:*

S.1 $[true]_i = true$ *and*

S.2 $[c \sqcup d]_i = [c]_i \sqcup [d]_i$ *for all* $c,d \in Con$.

We refer to S.1 as *emptiness*, intuitively signifying that empty spaces amounts to no information at all. S.2 is referred to as $\sqcup$-*distribution*, meaning that spaces distribute over the joining of information. A derived property of S.2 is monotonicity of spaces; for all $i = 1 \ldots n$,

S.3 if $c \sqsubseteq d$ then $[c]_i \sqsubseteq [d]_i$ for all $c,d \in Con$.

The intuition behind S.3 is that the structure of the information (w.r.t. $\sqsubseteq$) is preserved inside spaces. We now define extrusion in spatial constraint systems.

Definition 2.4 (scse)**.** *An n-agent* spatial constraint system with extrusion *is a scs equipped with n self-maps* $\uparrow_1, \ldots, \uparrow_n$ *(called extrusion functions) over its set of elements. Additionally, for each map* $\uparrow_i : Con \to Con$*:*

E.1 $[\uparrow_i c]_i = c$ *for all* $c \in Con$.

E.1 means that the i-th extrusion function is the right inverse of the i-th space function. One might also require that the extrusion function satisfy duals of S.1 and S.2:

E.2 $\uparrow_i(true) = true$, and

E.3 $\uparrow_i(c \sqcup d) = \uparrow_i c \sqcup \uparrow_i d$ for all $c,d \in Con$.

It is not unreasonable to suppose that the extrusion function might be a semantical interpretation of an operation that satisfies emptiness and $\sqcup$-distribution.

Interface and usage. Table 2 exposes part of the interfaces to the `scs` and the `scs-e` modules. The interfaces are similar in that both offer methods to retrieve, set and modify the space/extrusion functions. However, with module `scs-e` it is possible to evolve a scs into a scse according to a choice function that automatically maps the extrusion functions. There are four choice functions implemented ; i. *infima*, ii. *suprema*, iii. *manual* and iv. *random*.

```
cs<int> CS( 0, 5 ); // true = 0, false = 5
CS.add_element( 1 ); // 0 <= 1 <= 5
CS.add_element( 2 ); // 0 <= 2 <= 5
CS.add_element( 3, {1, 2} ); // 1,2 <= 3 <= 5
CS.add_element( 4, {2} ); // 2 <= 4 <= 5
CS.lub( {1, 2} ); // lub(1,2) = 3
CS.glb( {2, 3, 4} ); // glb(2,3,4) = 2
CS.imp( 2, 3 ); // 2 -> 3 = 1
CS.is_distributive(); // cs IS distributive.
```

Figure 3. Snippet using the `cs` module

Table 2. Interface to `scs` and `scs-e`

Method	Desc.	Symbol
`T s( int i, T c ) / T e( int i, T c )`	space/extrusion functions	$[c]_i, \uparrow_i c$
`vector<T> s_inv( int i, T c )`	inverse of space function	$[c]_i^{-1}$
`vector<T> e_inv( int i, T c )`	inverse of extrusion function	$\uparrow_i^{-1} c$
`s_map( int i, T c, vector<T> elems )`	mapping of space function	$[elems]_i = c$
`e_map( int i, T c, vector<T> elems )`	mapping of extrusion function	$\uparrow_i elems = c$

The choice function *manual* expects the user to set the extrusion function using `e_map` after the creation of the scse. The choice function *random* maps each constraint $c \in Con$ to a random element of its space function pre-image (i.e. $[c]_i^{-1}$). Choice functions *infima* and *suprema* map each constraint to the greatest lower bound and least upper bound appropriately of its space function pre-image. Mathematically speaking, for each $c \in Con$ we have $\uparrow_i c = \bigsqcap [c]_i^{-1}$ and $\uparrow_i c = \bigsqcup [c]_i^{-1}$ when using the *infima* and *sumprema* choice functions respectively. Choice functions *random* and *manual* do not necessarily satisfy E.1 while *infima* and *sumprema* do, moreover the choice function *infima* satisfies E.2 and E.3 [2].

Properties of scs and scs-e. Several properties of the space/extrusion functions might be desired or needed for correct functioning (e.g. E.1 as mentioned above). Both modules offer property checking via the methods `s_properties` and `e_properties`. One can verify standard properties like surjectivity (this implies the existence of an inverse function), $\sqcup$-distributivity (i.e. S.2, E.3) and inversion (i.e. E.1) among others.

The snippet in Figure 4 creates a scs out of the cs created in Figure 3 and maps some of its elements. Next, it creates a scs-e with this scs. Here, the parameter `EC_SUPREMA` corresponds to the choice function *suprema*.

```
scs<int> SCS( CS, 1 ); // 1-agent scs, s_1(0) = 0
↪  mapped at creation
SCS.s_map( 1, 1, {1, 2, 3} ); // s_1({1,2,3}) = 1
SCS.s_map( 1, 4, {4} ); // s(4) = 4
SCS.s_map( 1, 5, {5} ); // s(5) = 5
SCS.s( 1, 4 ); // 4
SCS.s_inv( 1, 1 ); // {1,2,3}

scse<int> SCSE( SCS, EC_SUPREMA ); // e_1(c) =
↪  lub(s_1_inv(c))
SCSE.e( 1, 1 ); // lub(s_1_inv(1)) = lub({1,2,3})
↪  = 3
SCSE.e_inv( 1, 5 ); //  2,3,5
SCSE.e_map( 1, 2, {2} ); // e(2) = 2
SCSE.e_properties( 1, EP_RIGHT_INVERSE_S ); //
↪  e_1 is NOT the right inverse of s_1
```

Figure 4. Snippet using the `scs` and `scse` modules

2.1. Complexity

We now turn our attention to the details of time complexity (see Table 3 for a complete chart). Implementation of lattices operators, and by extension those of constraint systems, might yield considerable time complexities due to the potentially large number of elements in the cs. We discuss the most critical cases here, those of methods `leq`, `glb`, `lub` and `imp`. Recall that posets were implemented using a `BGL` graph.

`leq`. The result of $c \sqsubseteq d$ can be given in constant time provided this is calculated in advanced. We achieve this by performing a transitive closure on the poset relation whenever an element is added (i.e. method `add_element`). This transitive closure is performed using the `BGL` method `boost::transitive_closure`.

`glb` and `lub`. We take advantage of the fact that posets in cs are always in transitive closure to lower the complexity of calculating meets and joins. The meet and join of a set of elements S are defined as $glb(S) = max(S^l)$ and $lub(S) = min(S^u)$ respectively, where S^l (lower bounds of S) is defined as the set $\{e \mid \forall_{s \in S}\ e \sqsubseteq s\}$ and S^u (upper bounds of S) as the set $\{e \mid \forall_{s \in S}\ e \sqsupseteq s\}$ [6]. Moreover, S^l and S^u can be calculated in consant time with `BGL` methods `boost::inv_adjacent_vertices` and `boost::adjacent_vertices`. Calculating $max(S^l)$ and $min(S^u)$ then boils down to finding a minimum value as the next proposition shows. Corollary 2.1 follows from Proposition 2.1.

Proposition 2.1. $max(S^l) = \underset{s_i \in S^l}{arg\,min} |s_i{}^u|$

Proof. Suppose not, then $s_k = max(S^l)$, $s_j = \underset{s_i \in S^l}{arg\,min} |s_i{}^u|$ and $s_j \sqsubset s_k$ because the maximal element is unique ($\sqsubseteq$ is antisymmetric by Definition 2.1). Furthermore, $s_k{}^u \subset s_j{}^u$ because $\sqsubseteq$ is transitive. Consequently $|s_k{}^u| < |s_j{}^u|$, a contradiction. □

Table 3. Worst-case complexity of methods, n means # of elements in the cs

Method	Complexity	Method	Complexity
`add_element`	$\mathcal{O}(n^3)$	`s, e`	$\mathcal{O}(1)$
`leq`	$\mathcal{O}(1)$	`s_inv, e_inv`	$\mathcal{O}(n)$
`glb, lub`	$\mathcal{O}(n^2)$	`s_map, e_map`	$\mathcal{O}(1)$
`imp`	$\mathcal{O}(n^3)$	`s_property`	$\mathcal{O}(n^2)$
`is_distributive`	$\mathcal{O}(n^3)$	`e_property`	$\mathcal{O}(n^2)$

Corollary 2.1. $min(S^u) = \underset{s_i \in S^u}{arg\,min} |{s_i}^l|$

`imp`. Recall that $c \to d = \bigsqcap S$ where $S = \{e \mid c \sqcup e \sqsupseteq d\}$, we lower the complexity by characterizing S. When $c \sqsupseteq d$ we have that $S = Con$, thereby $c \to d = \bigsqcap Con = true$. When $c \sqsupseteq d$ is not the case, it is easy to show that $d^u \subseteq S$, whereby $\bigsqcap d^u = d$, therefore we can safely omit all elements of d^u from S, except d (due to associativity of $\sqcap$).

Moreover, some elements need not be tested when calculating S. A particular common case is the negation (i.e. $d = false$), the elements of the set $(c^u \backslash \{false\})^l$ are never in S. The next proposition proves this.

Proposition 2.2. *$S' \cap S = \emptyset$ where $S = \{e \mid c \sqcup e \sqsupseteq false\}$ and $S' = (c^u \backslash \{false\})^l$.*

Proof. If $c = false$ then $S' = \emptyset$, thus the proposition is trivially true. If $c \neq true$ we prove that if $a \in S'$ then $a \notin S$. Suppose not, then $a \in S'$, meaning that $\exists a' \in c^u \backslash \{false\}$ and $a \sqsubseteq a'$. Furthermore $c \sqsubseteq a' \sqsubset false$ and $a \sqsubseteq a'$. We can show that $c \sqcup a \sqsubseteq a'$ and by transitivity we deduce that $c \sqcup a \sqsubset false$. Furthermore $a \in S$ (by supposition), meaning that $c \sqcup a \sqsupseteq false$, a contradiction. □

3. Semantical description of Social Network Behaviors

Boolean Algebras. Adopting D-SPACES for constructing proof of concepts using constraint systems with extrusion is feasible. To achieve of this, the module `ba` is offered as an implementation of powerset boolean algebras (ba). In this module, a constraint system is built automatically from a powerset lattice which in turn is constructed from a set of elements called *atoms*. The atoms represent the indivisible bits that make up the information in the constraint system, moreover, a powerset lattice is complete and distributive by construction [6].

Given a set of atoms A, a powerset ba is a specific case of a scse where $Con = \mathcal{P}(A)$, $\sqcup = \cup$ (or $\cap$ if the lattice is inverted), $\sqcap = \cap$ (or $\cup$ if inverted), $true = \emptyset$ (or A if inverted) and $false = A$ (or $\emptyset$ if inverted). Additionally, a boolean algebra is equipped with a complementation operation (i.e. c') that we calculate by using the pseudo-complement[4] defined in Section 2.

```
ba<char> BA( {'c', 'a', 'b'}, 2, true );
// space function
auto s = [] (int i, set<char> e, set<char> atoms)
↪  {
  switch( i ) {
    case 1: // s_1(c) = c
      return e;
    case 2: // s_2(c) = A \ {c}
      return set_difference( atoms.begin(),
↪  atoms.end(), e.begin(), e.end() );
  }
};
BA.map_s( s, EC_INFIMA ); // e_n(c) =
↪  glb(s_n_inv(c))
BA.m_scse.is_distributibe(); // All powerset
↪  lattices are distributive
```

Figure 5. Snippet using the `ba` module

Space and extrusion functions are defined programmatically using `c++11` lambda functions. Because the powerset ba is also a scse, the module also exposes all the functionality of the constraint systems discussed up until this point. The code example in Figure 5 creates a two-agent powerset boolean algebra and automatically maps the extrusion functions as the *infima* choice of user-given space functions.

3.1. A Tagged Social Network

We proceed to illustrate an application of constraint systems with extrusion as semantics for epistemic behaviors in social networks. We define a social network of *comments* that are tagged according to their content, the tags used are the following:

`h`: Personal. `n`: News.

`p`: Political. `s`: Sports.

`r`: Religious.

We create a powerset boolean algebra to represent

[4] In powerset lattices, the complement and the pseudo-complement are equivalent.

the social network, its set of atoms being the above tags. Additionally, there are three users in the social network represented as the three agents of the constraint system:

```
ba<char> ReseauSocial( { 'h', 'p', 'r', 'n', 's' },
  ↪  3 );
```

Users in this social network are allowed to have their own set of beliefs inside their spaces (e.g. walls) and make opinions about the existing information (e.g. posts). We intend to use a boolean algebra to calculate the semantic meaning of scenarios where these opinions and beliefs exist together. For this we express the epistemic behaviors using the logical language of belief and utterance BU_n [2]:

$$F := t \mid F \wedge F' \mid \neg F \mid B_i(F) \mid U_i(F)$$

where $i = 1 \ldots n$. In BU_n, a comment F can be a tag t, a conjunction of comments, a negation of a comment, a user belief (i.e. $B_i(F)$ stands for "user i believes F") and a user utterance (i.e. $U_i(F)$ stands for "user i utters F").

We assign to each user a *profile* that dictates how he *believes* and *utters* comments. User 1 is a political person and at the same time discreet of his personal life, user 2 has a very religious character while being apolitical and finally user 3 is an objetive individual. We emulate their belief profiles by applying the next lambda function as the space function of the social network:

```
auto belief_func = [] (int agent, std::set<char>
  ↪  comment, std::set<char> tags) {
       std::set<char> belief;
       switch( agent ) {
           case 1:
               belief = comment;
               if( belief.find( 'n' ) !=
  ↪  belief.end() )
                   belief.insert( 'p' );
               break;
           case 2:
               belief = comment;
               if( belief.find( 'h' ) !=
  ↪  belief.end() )
                   belief.insert( 'r' );
               break;
           case 3:
               belief = comment;
               break;
       }
       return belief;
   };
   ReseauSocial.map_s( belief_func );
```

Notice how user 1 inserts in every news a political aspect, while user 2 give to his personal comments a religious interpretation. User 3 is objective and interprets the comment unchanged. We also create a lambda function to code the uttering profiles:

```
auto utterance_func = [] (int agent, std::set<char>
  ↪  comment, std::set<char> tags) {
       std::set<char> utterance;
       switch( agent ) {
           case 1:
               utterance = comment;
               utterance.erase( 'h' );
               if( utterance.find( 'n' ) !=
  ↪  utterance.end() )
                   utterance.insert( 'p' );
               break;
           case 2:
               utterance = comment;
               utterance.erase( 'p' );
               break;
           case 3:
               utterance = comment;
               break;
       }
       return utterance;
   };
   ReseauSocial.map_e( utterance_func );
```

In this case user 1 inserts a political angle in every news but removes any personal detail from a comment. User 2 removes the political aspect in the comment and user 3 remains objective. To interpret statements of epistemic behavior in the social network we inductively give semantics to the language of belief and utterance using constraint systems with extrusion. We define a function $[\![\cdot]\!] : F \mapsto Con$ that maps a statement from BU_n to a constraint of `ReseauSocial`.

$$\begin{aligned}
[\![t]\!] &= \{\mathtt{t}\} \\
[\![F \wedge F]\!] &= [\![F]\!] \sqcup [\![F]\!] \\
[\![\neg F]\!] &= \sim [\![F]\!] \\
[\![B_i(F)]\!] &= [[\![F]\!]]_i \\
[\![U_i(F)]\!] &= \uparrow_i [\![F]\!]
\end{aligned}$$

A tag is semantically interpreted as a set containing the tag, the conjunction of comments is interpreted as their join, the negation as the pseudo-complement and the belief and utterance actions as the space and extrusion operators respectively. We now present some epistemic scenarios where we use the boolean algebra representing the social network to calculate their semantical meaning. As a first case, we want to model the belief of user 2 of a news comment that user 1 believes true and utters to him:

$$B_2(B_1(news \sqcup U_1(news)))$$

We encode this scenario in the social network as follows:

```
ReseauSocial.m_scse.s( 2, ReseauSocial.m_scse.s( 1,
  ↪  ReseauSocial.m_scse.lub( { {'n'},
  ↪  ReseauSocial.m_scse.e( 1, {'n'} ) } ) ) );
```

The semantical result of the above statement is $\{\mathtt{'n','p'}\}$ indicating that the subjective (possible wrong) political interpretation of the news from user 1 was also picked up by user 2. Next, we model an scenario where user 1 is a friend of user 2 and he comments on their mutual personal activities (a sport activity denoted here by *personal*). For this, user 1 verifies if the

activity is common to them, and utters such activity as interpreted by user 2:

$$B_1(personal) \rightarrow U_1(B_2(personal \sqcup sports)) \sqcup B_1(personal)$$

The scenario is encoded as follows:

```
ReseauSocial.m_scse.lub( { ReseauSocial.m_scse.imp(
  ↪  ReseauSocial.m_scse.s( 1, {'h'} ),
  ↪  ReseauSocial.m_scse.e( 1,
  ↪  ReseauSocial.m_scse.s( 2,
  ↪  ReseauSocial.m_scse.lub( { { 'h', 's'} } ) ) )
  ↪  ), ReseauSocial.m_scse.s( 1, {'h'} ) } );
```

The result here, $\{$`'h','r','s'`$\}$, shows that the semantical interpretation mixes religious and sport tags in the same scenario. Such configurations could be considered potentially problematic and politically incorrect for a moderator of the social network. For the last scenario we want to model user 3 as an active liar where he intentionally utters to the other users news he regards as untrue. User 2 however already believes the news to be untrue:

$$B_3(\neg news \sqcup \neg news \rightarrow U_3(B_1(news) \sqcup B_2(news))) \sqcup B_2(\neg news)$$

```
ReseauSocial.m_scse.lub( { ReseauSocial.m_scse.s( 3,
  ↪  ReseauSocial.m_scse.lub( {
  ↪  ReseauSocial.m_scse.imp(
  ↪  ReseauSocial.m_scse.imp( {'n'},
  ↪  {ReseauSocial.m_scse.lub()} ),
  ↪  ReseauSocial.m_scse.e( 3,
  ↪  ReseauSocial.m_scse.lub( {
  ↪  ReseauSocial.m_scse.s( 1, {'n'} ),
  ↪  ReseauSocial.m_scse.s( 2, {'n'} ) } ) ) ),
  ↪  ReseauSocial.m_scse.imp( {'n'},
  ↪  {ReseauSocial.m_scse.lub()} ) } ) ),
  ↪  ReseauSocial.m_scse.s( 1,
  ↪  ReseauSocial.m_scse.imp( {'n'},
  ↪  {ReseauSocial.m_scse.lub()} ) ) } );
```

The semantical result is $\{$`'h','n','p','r','s'`$\}$ which is the top element of the constraint system. This can be interpreted as *false* due to the inconsistency generated in the beliefs of agent 2 after the news is uttered to him by agent 3.

4. Conclusions and Future Work

We presented D-SPACES, an implementation of constraint systems with space and extrusion operators for semantically describing information structured in spaces. We covered the different definitions of constraint systems as well as an implication operator to increase expressivity. Additionally, we documented the different methods in the implementation to verify conditions in the constraint systems that might be desired for certain properties to hold. To implement the ordering relation of a constraint system we used the `BGL`'s implementation of graphs. This, together with some mathematical results, allowed us to work on the complexity of the cs operators.

As a way to code proof of concepts on scse we introduced a module to create powerset boolean algebras (a specific case of scse) with space and extrusion functions specified as lambda functions. Furthermore, we illustrated the use of scse with a semantical interpretation of an epistemic language of belief and utterance. Here, we created a small social network as a powerset ba and discussed the resulting semantical interpretations of different epistemic behaviors described in the aforementioned language.

As future endeavors we plan to implement more significant cases of scse and support more data types. This will allow for more description languages to be interpreted easier and quicker. We would also like to see support for removing elements, as this, together with the `add_element` method, would allow to interactively manipulate a scse and give meaning to a constantly changing structure of information. Finally, we envisage that results from an interpretation of a language can be coupled with other tools to perform verification and/or detection of desired/undesired features.

References

[1] S. Knight, C. Palamidessi, P. Panangaden, and F. D. Valencia, "Spatial and epistemic modalities in constraint-based process calculi," in *Proceedings of the 23rd International Conference on Concurrency Theory, CONCUR 2012*, pp. 317–332, Springer, 2012.

[2] S. Haar, S. Perchy, C. Rueda, and F. D. Valencia, "An algebraic view of space/belief and extrusion/utterance for concurrency/epistemic logic," in *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming, Siena, Italy, July 14-16, 2015*, pp. 161–172, 2015.

[3] R. Fagin, J. Y. Halpern, Y. Moses, and M. Y. Vardi, *Reasoning about knowledge*. MIT press Cambridge, 4th ed., 1995.

[4] H. Van Ditmarsch, J. Van Eijck, F. Sietsma, and Y. Wang, "On the logic of lying," in *Games, actions and social software*, pp. 41–72, Springer, 2012.

[5] V. A. Saraswat, M. Rinard, and P. Panangaden, "Semantic foundations of concurrent constraint programming," in *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 333–352, 1991.

[6] B. A. Davey and H. A. Priestley, *Introduction to lattices and order*. Cambridge university press, 2nd ed., 2002.

[7] S. Abramsky and A. Jung, "Domain theory," *Handbook of logic in computer science*, pp. 1–77, 1994.

[8] S. Vickers, *Topology via logic*. Cambridge University Press, 1st ed., 1996.