Declarative Framework for Semantical Interpretations of Structured Information - An Applicative Approach

Stefan Haar, Salim Perchy, Frank Valencia

DECLARATIVE FRAMEWORK FOR SEMANTICAL INTERPRETATIONS OF STRUCTURED INFORMATION. AN APPLICATIVE APPROACH

STEFAN HAAR

LSV, École Normale Supérieure de Cachan, 61 Avenue du Président Wilson

Cachan, Île-de-France 94235, France

stefan.haar@inria.fr

`http://www.lsv.fr/~haar/`

SALIM PERCHY

Loria, INRIA Grand-Est, 615 rue du Jardin Botanique

Villers-lès-Nancy, Meurthe-et-Moselle 54600, France

yamil-salim.perchy@inria.fr

`http://www.lix.polytechnique.fr/~perchy/`

FRANK VALENCIA

LIX, École Polytechnique de Paris, 1 rue Honoré d'Estienne d'Orves

Palaiseau, Île-de-France, 91120, France

frank.valencia@polytechnique.fr

`http://www.lix.polytechnique.fr/~fvalenci/`

Abstract

We study the applicability of declarative models to encode and describe structured information by means of semantics. Specifically, we introduce 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 mainly give this information an epistemic and temporal interpretation and consider various *agents* as entities acting upon it. D-SPACES is coded as a `c++` library providing implementations of 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. Finally, we show the applicability of this framework with two examples; a scenario in the form of a social network where users post their beliefs and utter their opinions, and a semantical interpretation of a logical language to express time behaviors and properties.

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 [12, 15]. We aim to achieve a clear understanding of the concept of space and extrusion that enables us to study the meaning of information in these systems.

Initially, we focus on epistemic systems where we have agents *believing* information [9] and uttering opinions and lies [19]. 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 (the information) [18]. Later on, we demonstrate how the concepts of space and extrusion have a tight semantical relationship with those of the *future* and *past* operations in discrete time processes. We think this relationship applies to scenarios where time properties are critical to their correct execution and/or expected output. For this specific goal, we show how the proposed framework can create an accurate semantical description of temporal logics, that in turn are mature languages for specification of reactive systems.

Constraints can be viewed as assertions representing partial information (e.g. $t < 50$ may stand for a certain temperature bellow 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 [12]. 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 that agent j believes c" respectively. Movement of information across spaces was introduced by means of the constraint operator $\uparrow$ called *extrusion* [10]. One can now conceive statements like $[\uparrow_i [c]_j]_i$ "agent i extrudes the information c to agent j" or epistemically as "agent i extrudes that agent j believes c". Space and extrusion functions may also be read as temporal assertions; $[c]$ can be interpreted as "c holds in the next instant" while $\uparrow c$ as "previously c was true".

The purpose of this work is to present the usability of the tool D-SPACES; an implementation of constraint systems with space and extrusion operators. We also use the tool to provide a semantic language for describing systems where information is structured in a hierarchy of spaces. D-SPACES[1] was conceived as a `c++` library heavily relying on the boost graph implementation[2] (`BGL`). It is thoroughly documented using `HeaderDoc`[3] and can be directly used in the OS X (XCode) and Linux (GCC+Make) environments. Usage on Windows depends upon

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

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

[3] `https://developer.apple.com/legacy/library/documentation/DeveloperTools/Conceptual/HeaderDoc/`

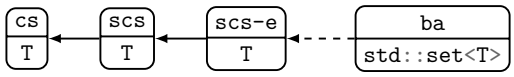

Figure 1: General class diagram

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 formally 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 and finally present powerset boolean algebras as a specific instantiation of our model. In section 3 we use these algebras to give semantics to a belief language describing a small social network based on tags. In addition to this, we present a semantical interpretation of the Linear Time Temporal Logic [16] language by way of the models studied here and show its potential use for asserting time behaviors and/or results. Finally, Section 4 offers some concluding remarks and future endeavors regarding the methodology exposed here and the tool.

2 Implementing Space and Extrusion in Constraint Systems

Constraint systems are declarative formalisms which specify partial information that programs (processes) may act upon [14]. Intuitively, constraint systems build a specific structure of information (that of a poset) and define useful operations between the different elements that make up the totality of the information. Likewise, the notion of computational space and the movement of the information therein may be extended to constraint systems by means of space and extrusion functions [10]. In previous works, formal models describing the concept of space were generally treated from a theoretical standpoint and much of their results are mathematical in nature [6, 5]. To demonstrate their applicability to real computational problems, the authors have developed D-SPACES, a tool implementing constraints systems with extrusion.

We begin by describing the class hierarchy of D-SPACES (Fig. 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`, `std::pair<std::string,int>>` and containers `std::vector` and `std::set` of these same types. We continue by introducing the necessary mathematical concepts to describe the first component; the `cs` module.

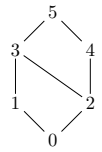

Figure 2: A Poset

2.1 Flat Constraint Systems

We first formalize the concept of *constraint system*. A basic background in domain theory is presupposed [8, 1].

Definition 2.1 (Lattice)**.** *A* lattice *is a partially ordered set (poset)* $(Con, \sqsubseteq)$ *where for each* $c, d \in Con$ *we define;*

(i) $c \sqcap d$ *as the maximal element* e *w.r.t.* $\sqsubseteq$ *s.t.* $e \sqsubseteq c$ *and* $e \sqsubseteq d$ *(read as the* meet *of* c *and* d*) and,*

(ii) $c \sqcup d$ *as the minimal element* e *w.r.t.* $\sqsubseteq$ *s.t.* $c \sqsubseteq e$ *and* $d \sqsubseteq e$ *(read as the* join *of* c *and* d*).*

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 Fig. 2 is the Hasse diagram of its underlying poset.

Definition 2.2 (cs)**.** *A* constraint system *[18] is a complete lattice, that is, a lattice where the meet and join operations are defined for every subset of the set 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 the global join $\top$). 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 Fig. 2, *true* is the element 0 and *false* is the element 5, needlessly to say, *false* entails all the elements of the cs.

Example 2.1 (Herbrand Constraint System)**.** *The Herbrand cs [4, 18] captures* syntactic *equality between terms* $t, t', \ldots$ *built from a first-order alphabet* $\mathcal{L}$ *with variables* $x, y, \ldots$*, function symbols, and equality* $=$*. The elements are (equivalent classes of) sets of equalities over the terms of* $\mathcal{L}$*: E.g.,* $\{x = t, y = t\}$ *is an element. The relation* $c \sqsubseteq d$ *holds if the equalities in* c *follow from those in* d*: E.g.,* $\{x = y\} \sqsubseteq \{x = t, y = t\}$*. The top element false is the set of all (possibly*

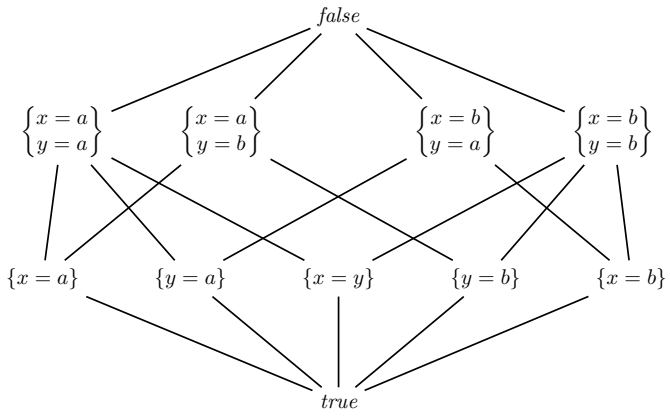

Figure 3: A Herbrand constraint system

inconsistent) term equalities in $\mathcal{L}$ and true is (the equivalence class of) the empty set. The join operation is the (equivalence class of) set union. Figure 3 is the hasse diagram of a Herbrand cs with variables $\{x, y\}$ and constants $\{a, b\}$ with $a \neq b$. $\square$

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* [20] and it additionally allows us to encode the pseudo-complement of a constraint as $\sim c = c \to \mathit{false}$. Pseudo-complements do not necessarily comply with the law of the excluded middle and the above definition only works as an implication if the cs is distributed, that is if for every $a, b, c \in \mathit{Con}$ we have that:

$$a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c) \tag{2.1}$$

Notice that, by the $\mathbf{M}_3$-$\mathbf{N}_5$ theorem [8], Herbrand constraint systems are not distributed (i.e. their underlying lattices are sub-lattices of the $\mathbf{N}_5$ lattice).

2.1.1 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

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	$c \to d$

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.

2.1.2 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 $c, d \in Con$ [11]. In D-SPACES distributivity can be checked with the boolean method `CS.is_distributive()`.

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

2.2 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 Con. 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 an empty space 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 $c, d \in Con$.

```
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 4: Snippet using the `cs` module

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 Con. 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.

Example 2.2 (Structured information)**.** *Let us consider the following computational setting:*

$$[c]_i \sqcup [\uparrow_j([c \to d]_i) \sqcup e]_j \tag{2.2}$$

Equation 2.2 specifies the sending of information d from agent j to agent i. This action is conditioned by the presence of information c in the space of agent i. Indeed, with the help of S.2, E.1 and modus ponens we can derive the expected result as follows:

$$\begin{aligned}
&[c]_i \sqcup [\uparrow_j([c \to d]_i) \sqcup e]_j \\
&= [c]_i \sqcup [\uparrow_j([c \to d]_i)]_j \sqcup [e]_j && \textit{(S.2)} \\
&= [c]_i \sqcup [c \to d]_i \sqcup [e]_j && \textit{(E.1)} \\
&= [c \sqcup c \to d]_i \sqcup [e]_j && \textit{(S.2)} \\
&= [c \sqcup d]_i \sqcup [e]_j && \textit{(MP)}
\end{aligned}$$

This derivation corresponds to the movement of a piece of (partial) information among a hierarchy of spaces that structures the (total) information in the system. □

2.2.1 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*.

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 [11].

2.2.2 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 Fig. 5 creates a scs out of the cs created in Fig. 4 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*.

2.3 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. We discuss the critical cases here, those of methods `leq`, `glb`, `lub` and `imp`. Recall that posets were implemented using a `BGL` graph.

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`	$\mathcal{O}(n^2)$

```
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 5: Snippet using the `scs` and `scse` modules

2.3.1 `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`.

2.3.2 `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 $\{e \mid \forall_{s \in S}\ e \sqsupseteq s\}$ [8].

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. □

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

2.3.3 `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 $\sqcap$).

Additionally, 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. □

2.4 Boolean Algebras

Adopting D-SPACES for constructing proof of concepts using constraint systems with extrusion is feasible. To achieve 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 [8].

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.

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 Fig. 6 creates a two-agent powerset boolean algebra and automatically maps the extrusion functions as the *infima* choice of user-given space functions.

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

```
ba<char> BA( {'c', 'a', 'b'}, 2, true ); // A = {a,b,c}
// 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 6: Snippet using the `ba` module

3 Applicability of Semantical Descriptions for Structured Information

We now focus on the applicability of constraint system semantics to relevant computational scenarios. In this section we provide two different settings, one of a social network with users capable of posting comments that are automatically tagged and another of a logical language that is mainly used to express time behaviors and/or properties in programs. Both settings have been and continue to be active subjects in the literature [2, 13].

On both cases the methodology remains the same: we first identify how the information is structured (that is, we instantiate the concept of space) and then we proceed to describe the information and its operations as a semantical language based on constraint systems. Naturally, this allows us to encode the setting's computational behaviors as D-SPACES methods that yield results for analysis or input to other computations.

3.1 A Tagged Social Network

As our first illustrative example, we use space and extrusion functions 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 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 walls (i.e. the concept of *space*) and make opinions/posts about the existing information (i.e. the concept of *extrusion*). We intend to use the 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 multi-agent language of belief and utterance BU_n [11]:

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

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 gives 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 of 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 {'n', 'p'} indicating that the subjective (possibly wrong) political interpretation of the news from user 1 was also picked up by user 2. Next, we model a scenario where users share beliefs (e.g. friendship) and user 1 comments on a mutual personal activity with agent 2 (a sport activity denoted here by *personal*). For this, user 1 verifies if the activity in question is common to them (i.e. $B_1(personal)$), 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 (due to the profile of agent 2). 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. For this scenario, we adopt a logical and epistemic description of a lie[19, 17]:

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

```
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.

3.2 A Time Description Language

Our next application of D-SPACES features the ability to express and verify time behaviors in computer programs. We focus on the following computational process as the main target of this application:

```
procedure PROGRAM(arg1, arg2)
    a ← 1
    b ← 4
    while a < b do
        if even(a) then
            a ← a + arg1
        end if
        if odd(a) then
            a ← a + arg2
        end if
    end while
    if a = b then
        c ← 1
    else
        c ← 0
    end if
end procedure
```

To describe such behaviors, we use an established language that is capable of expressing propositions with temporal characteristics. LTL (Linear Temporal Logic) is a logical language with time operators (contextually called *temporal modalities*) to describe temporal properties [16]. We write its syntax as follows:

$$F := v \mid F \wedge F' \mid F \vee F' \mid \bigcirc F \mid \ominus F$$

Here, a basic proposition v stands for a variable value, this relates to the variable assignation concept we see in normal computer programs (such as our target PROGRAM). Notice that LTL is a special case of a modal logic, which in turn is an extension of propositional logic [3], thus the last two operators are its temporal modalities. $\bigcirc F$ is read as "next F" and means that in the next time instant F is valid. $\ominus F$ on the other hand, read as "previous F", is interpreted as F being valid in the previous time instant. It is natural to conceptually tie the concept of *next* and *previous* to those of *space* and *extrusion* respectively.

The expressivity of LTL can be further expanded with additional temporal modalities that are directly related to the *next* and *previous* operations. We extend

the syntax with the following operators:

$$F := \cdots \mid \Box F \mid \boxminus F \mid \Diamond F \mid \diamondminus F$$

$\Box F$ and $\boxminus F$ are read as "henceforth F" and "always F" respectively. The former states that from now on F holds, while the latter states that F has always been true up to this point. $\Diamond F$ and $\diamondminus F$ are read as "eventually F" and "once F". The first one is interpreted as F being true at one point in the future and the second one as F being valid at one point in the past. These extra operations can be encoded with the first two modalities as follows:

$$\begin{aligned}
\Box F &:= F \wedge \bigcirc F \wedge \bigcirc \bigcirc F \wedge \bigcirc \bigcirc \bigcirc F \wedge \ldots \\
\boxminus F &:= F \wedge \ominus F \wedge \ominus \ominus F \wedge \ominus \ominus \ominus F \wedge \ldots \\
\Diamond F &=: F \vee \bigcirc F \vee \bigcirc \bigcirc F \vee \bigcirc \bigcirc \bigcirc F \vee \ldots \\
\diamondminus F &:= F \vee \ominus F \vee \ominus \ominus F \vee \ominus \ominus \ominus F \vee \ldots
\end{aligned}$$

The basic sentences (i.e., v) of the LTL language defined above are variable values. In contrast to the last application on social networks, we do not use a boolean algebras as our main constraint system structure. A much more apt structure for this scenario is a Herbrand constraint system (see Example 3). Programmatically, the creation of a Herbrand cs is very similar to that of the powerset of boolean algebras. The constraint system elements are possible valuations of the variables in the target program (e.g., Procedure Program). These elements are represented by sets of pairs (i.e., variable/valuation) in our example. Inconsistent elements (e.g., states containing variables with double valuations) are eliminated at construction time as they are never reached in a program. We omit the code for constructing Hebrand cs but the interested reader is referred to the demos of D-SPACES:

```
// Herbrand cs element type: set of (var, val) where var = val
typedef std::set<std::pair<std::string,int>> CS_TYPE;

int nRuns = 2;
std::vector<std::string> vars = {"a", "b", "c"};
std::pair<int,int> range( 0, 5 );
cs<CS_TYPE> CS = herbrand( vars, range ), nRuns );
scse<CS_TYPE> HCS(scs<CS_TYPE>( CS, nRuns ),
↪  scse<CS_TYPE>::E_CHOICE_FUNCTION::EC_MANUAL );
```

In Procedure Program we use variables a, b and c (`vars`) with a permitted value range from 0 to 5 (`range`). We use agents of the constraint systems to define different *runs* of the program with different argument values. The number of runs is determined by variable `nRuns`. Contrary to the social networks application where the space and extrusion functions were intentionally defined, here we specify them in a procedural fashion. For this, we create an evaluation function to encapsulate Procedure Program:

```
template<typename... ArgTypes>
void evaluate( scse<CS_TYPE>& hcs, int run, ArgTypes... args ) {
  CS_TYPE state; // Current state
  CS_TYPE next;  // Following state

  #define STEP( Var ) step( hcs, run, state, next, #Var, (Var) )

  // Target Program
  std::function<void(int,int)> program = [&hcs, run, &state, &next]
  ↪  ( int incr1, int incr2 ) {
    int a = 1;
    STEP(a);
    int b = 4;
    STEP(b);
    while( a < b ) {
      if( a % 2 == 0 )
        a = a + incr1;
      else
        a = a + incr2;
      STEP(a);
    }
    int c;
    if( a == b)
      c = 1;
    else
      c = 0;
    STEP(c);
  };
  program( args... );
  hcs.s_map( run, {next}, next );
}
```

Naturally, the framework should allow evaluation of a generic program (i.e., different number and/or type of arguments, different body, etc), Because of this, `evaluate` is a *variadic function* à la `C++11` where a parameter pack carries the different arguments for the target program. Moreover, the encapsulation of the target program inside the evaluation function allows for the capture of the constraint system and hides its handling from the body of the target program. In spite of this, the task of informing the cs of a variable assignment remains explicit by invoking a stepping function. The macro `STEP` allows to *stringify* a variable using the operator `#`.

```
// Stepping function: Updates the run on the Herbrand cs
void step( scse<CS_TYPE>& hcs, int run, // Herbrand cs and run id
           CS_TYPE& s,                  // Previous state
```

```
            CS_TYPE& n,                    // Following state
            std::string var, int val ) { // Variable assignment
            ↪  (i.e., var = val )
  // Remove previous assignment
  for( auto pair : n ) {
    if( pair.first == var )
    n.erase( pair );
  }
  n.insert( std::make_pair(var, val) );// Add assignment to state
  hcs.s_map( run, {s}, n );  // Update space (next) function
  hcs.e_map( run, {n}, s) ;  // Update extrusion (previous)
  ↪  function
  s = n;
}
```

The stepping function is basically in charge of progressively constructing the space and extrusion functions accordingly, also accounting for the run number in the Herbrand cs. It calculates the elements to be mapped taking into account the new assignment and then does the actual mapping. Notice that at the end of `evaluate`, the last state is mapped to itself with respect to the space function. This is crucial to the use of some LTL modalities such as $\Box F$.

We remark that, after the evaluation is done, the Hebrand cs resembles (in structure) a state machine, thus it also suffers from the state explosion problem [7]. Although it is out of the scope of this work, this caveat can be tackled with the possibility of an element removal method in the constraint system module (see Section 4). Additionally, Herbrand cs are not distributive, therefore the complement of an element is not consistent with its logical counterpart. This is expected because the complement of a variable valuation is not a natural (nor a trivial) operation in imperative languages. Nevertheless, this issue can be avoided if the structure is made distributive by requiring the introduction of elements that are logically inconsistent (i.e, double valuations), though they are never reached by the space or extrusion function.

We now proceed to inductively give a semantical description to every operator of the LTL language:

$$
\begin{aligned}
[\![v]\!]^s &= \begin{cases} \bot & \text{if } \{v\} \sqsubseteq s \\ \top & \text{otherwise} \end{cases} \\
[\![F \wedge F]\!]^s &= [\![F]\!]^s \sqcup [\![F]\!]^s \\
[\![F \vee F]\!]^s &= [\![F]\!]^s \sqcap [\![F]\!]^s \\
[\![\bigcirc F]\!] &= [\![F]\!]^{[s]} \\
[\![\ominus F]\!] &= [\![F]\!]^{\uparrow s}
\end{aligned}
$$

The semantical meaning of a basic proposition is either the top and bottom elements of the cs. The result is decided based on its *entailment* from an element

s that represents the current state of the target program. This of course ties to the semantic *principle of bivalence* that is present by a logical language like LTL. Disjunctions and conjunctions are semantically interpreted by, naturally, meets and joins respectively. The *next* and *previous* operations are mapped to a semantical evaluation on the space and extrusion valuation of the element s. Notice that space and extrusion functions are devoid of agent indexes for simplicity reasons, we expect every semantical sub-evaluation to be consistent on the same *run*.

We are now ready to express a time behavior in LTL using our constraint system and test its validity in a run of the target program and its underlying Herbrand cs. We start by testing if variable b is always 4 in the program (i.e., $\Box\, b = 4$).

```
int cRun = 1;
evaluate( HCS, cRun, 1, 2 ); // arg1 = 1, arg2 = 2

CS_TYPE proposition;
proposition.insert( std::make_pair("b", 4) ); // p: b == 4
CS_TYPE answer = HCS.glb();
CS_TYPE current_state;
CS_TYPE next_state = HCS.s( cRun, HCS.s( cRun, HCS.glb() ) ); // The
↪  state when variable b is created
do {
  current_state = next_state;
  answer = HCS.lub( { answer, ( HCS.leq( proposition, current_state
  ↪  ) ? HCS.glb() : HCS.lub() ) } ); // p<=s and p<=[s] and ...
  next_state = HCS.s( cRun, current_state );
} while( next_state != current_state ); // end of run
```

We perform a run of Procedure PROGRAM with arguments 1 and 2, then create the proposition $b = 4$. Next, we semantically evaluate the *henceforth* operation by consecutive conjunctions of the *next* operation until the last resulting element of the evaluation. It is important to be aware that a query on the variable `b` only makes sense at or after the point where it was created, not before, hence we start evaluating $\Box\, b = 4$ just there. After computing the LTL expression, `answer` ends up being the bottom element (i.e., $\bot$) which semantically is interpreted as $\Box\, b = 4$ being valid.

We test a second LTL expression that checks if variable `c` is, at any point, equal to 1 (i.e., $\Diamond\, c = 1$). We perform this by using consecutive disjunctions starting at the initial element (i.e., $\bot$).

```
proposition.clear();
proposition.insert( std::make_pair("c", 1) ); // p: c == 1
answer = HCS.lub();
next_state = HCS.glb(); // Initial state of execution
do {
  current_state = next_state;
```

```
  answer = HCS.glb( { answer, ( HCS.leq( proposition, current_state
  ↪  ) ? HCS.glb() : HCS.lub() ) } ); // p<=s or p<=[s] or ...
  next_state = HCS.s( cRun, current_state );
} while( next_state != current_state ); // end of run
```

As expected, `answer` evaluates to the top element (i.e., $\Diamond\, c = 1$ is not true). This is due to the fact that `arg2` is 2. If we make a second run where `arg2` is 1 we can reevaluate the LTL sentence on the specific run.

```
cRun = 2;
evaluate( HCS, cRun, 1, 1 ); // program arguments: 1, 1

answer = HCS.lub();
next_state = HCS.glb(); // Initial state of execution
do {
  current_state = next_state;
  answer = HCS.glb( { answer, ( HCS.leq( proposition, current_state
  ↪  ) ? HCS.glb() : HCS.lub() ) } ); // p<=s or p<=[s] or ...
  next_state = HCS.s( cRun, current_state );
} while( next_state != current_state ); // end of run
```

In this case, `c` is eventually valued at 1, therefore $\Diamond\, c = 1$ is true in the second run of Procedure Program.

4 Conclusions and Future Work

We presented a declarative framework for semantically interpreting structured information. To show its applicability, we developed 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 declarative semantics with two scenarios. In one we constructed a declarative interpretation of an epistemic language of belief and utterance. We then used this interpretation to create a small social network as a powerset ba. Additionally we discussed the resulting semantical interpretations of different epistemic behaviors described in the language mentioned above. Another illustration of the use of our framework came as a semantical description of a logical language that expresses

time behaviors and properties. Here we looked over different logical expressions that describe information structured over a discrete timeline.

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 and a widened applicability on the type of computational scenarios that D-SPACES can tackle. 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. We believe that permanent mutating hierarchies of information are also of great significance in the possible applications of declarative semantics. 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.

Acknowledgments

This work has been partially supported by the Colciencias project 125171250031 CLASSIC, and 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).

References

[1] Samson Abramsky and Achim Jung. Domain theory. *Handbook of logic in computer science*, pages 1–77, 1994.

[2] Alexandru Baltag, Zoé Christoff, Rasmus K Rendsvig, and Sonja Smets. Dynamic epistemic logics of diffusion and prediction in social networks. *Draftpaper, April*, 2015.

[3] Patrick Blackburn, Maarten De Rijke, and Yde Venema. *Modal Logic*. Cambridge University Press, 1st edition, 2002.

[4] Frank S. Boer, Alessandra Di Pierro, and Catuscia Palamidessi. Nondeterminism and infinite computations in constraint programming. *Theoretical Computer Science*, pages 37–78, 1995.

[5] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part ii). In *Proceedings of the 13th International Conference of Concurrency Theory, CONCUR 2002*, pages 209–225, 2002.

[6] Luís Caires and Luca Cardelli. A spatial logic for concurrency (part i). *Information and Computation*, pages 194–235, 2003.

[7] Edmund M. Clarke and Orna Grumberg. Avoiding the state explosion problem in temporal logic model checking. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 294–303. ACM, 1987.

[8] Brian A Davey and Hilary A Priestley. *Introduction to lattices and order*. Cambridge university press, 2nd edition, 2002.

[9] Ronald Fagin, Joseph Y Halpern, Yoram Moses, and Moshe Y Vardi. *Reasoning about knowledge*. MIT press Cambridge, 4th edition, 1995.

[10] Michell Guzman, Stefan Haar, Salim Perchy, Camilo Rueda, and Frank D. Valencia. Belief, knowledge, lies and other utterances in an algebra for space and extrusion. *Journal of Logical and Algebraic Methods in Programming, JLAMP*, 86:107–133, 2017.

[11] Stefan Haar, Salim Perchy, Camilo Rueda, and Frank D. Valencia. An algebraic view of space/belief and extrusion/utterance for concurrency/epistemic logic. In *Proceedings of the 17th ACM SIGPLAN International Symposium on Principles and Practice of Declarative Programming, PPDP 2015*, pages 161–172, 2015.

[12] Sophia Knight, Catuscia Palamidessi, Prakash Panangaden, and Frank D Valencia. Spatial and epistemic modalities in constraint-based process calculi. In *Proceedings of the 23rd International Conference on Concurrency Theory, CONCUR 2012*, pages 317–332. Springer, 2012.

[13] Zohar Manna and Amir Pnueli. *Temporal Verification of Reactive Systems: Safety*. Springer Science & Business Media, 2012.

[14] Prakash Panangaden, Vijay Saraswat, Philip J Scott, and RAG Seely. A hyperdoctrinal view of concurrent constraint programming. In *Workshop of Semantics: Foundations and Applications, REX*, pages 457–476. Springer, 1993.

[15] Salim Perchy and Frank D. Valencia. Opinions and beliefs as constraint system operators. In *Technical Communications of the 31st International Conference on Logic Programming, ICLP 2015*, 2015.

[16] Amir Pnueli and Zohar Manna. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer, 1992.

[17] Chiaki Sakama, Martin Caminada, and Andreas Herzig. A logical account of lying. In *Proceeedings of the 12th European Conference of Logics in Artificial, JELIA 2010*, pages 286–299. Springer, 2010.

[18] Vijay A Saraswat, Martin Rinard, and Prakash Panangaden. Semantic foundations of concurrent constraint programming. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 333–352, 1991.

[19] Hans Van Ditmarsch, Jan Van Eijck, Floor Sietsma, and Yanjing Wang. On the logic of lying. In *Games, actions and social software*, pages 41–72. Springer, 2012.

[20] Steven Vickers. *Topology via logic*. Cambridge University Press, 1st edition, 1996.