D-SPACES: Simulating Distributed Information Using Constraint Systems and Order Theory

Stefan Haar, Salim Perchy, Frank Valencia

D-SPACES: Simulating Distributed Information Using Constraint Systems and Order Theory*

Stefan HAAR[1], Salim PERCHY[2], and Frank VALENCIA[3]

[1] CNRS, LSV École Normale Supérieur - Cachan
stefan.haar@inria.fr
[2] INRIA, LIX École Polytechnique - Saclay
yamil-salim.perchy@inria.fr
[3] CNRS, LIX École Polytechnique - Saclay and Universidad Javeriana - Cali
frank.valencia@polytechnique.fr

Abstract. This paper introduces D-SPACES, a tool for simulating constraint systems with space and extrusion operators. These formalisms are algebraic structures for modeling the concept of space as an operation over elements (the information). Furthermore, it is possible to specify movement of information from one space to another, either belonging to a different entity (the agents) or somewhere else in the space hierarchy. This tool is coded as a `c++11` library providing implementations for; a constraint system (cs), a cs with space functions (scs), and a scs with extrusion functions (scs-e). The interfaces to access each implementation are minimal and thoroughly documented. They also offer property-checking methods for verifying conditions on the cs and functions that are cumbersome to check by hand. Finally, D-SPACES provides an implementation of a specific but common type of a scs-e (a boolean algebra) for easy of access and proof of concept regarding these cs.

1 Motivation and Introduction

Systems where data moves across a given structure of information are now commonplace. Applications like social networks, forums, or any other that organizes its information in a defined hierarchy are among these systems. In practice, the nature of this information can be reviews, opinions, news, etc., which in turn belong to a certain entity, e.g. users, agents, applications. This relation of ownership can be conceptualized as *space*, thus a clear understanding of information in spaces and its movement across them is pertinent to the study of these systems.

Treating information as *knowledge* is one option. Here, we have agents *knowing* facts [1] and *uttering* opinions and lies [2]. However, to attain a sufficiently generic concept of space we study declarative formalisms. More specifically we use *constraint systems* (cs), algebraic structures that operate on elements called

* 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)

constraints (its representation of information) [3]. Constraints can be view as assertions representing partial information (e.g. $t < 50$), making cs ideal to model and manipulate data scattered throughout a hierarchy of spaces.

The characterization of a space operation $[\cdot]$ in constraint systems was developed in [4]. Assertions like $[c]_i$ "information c is present inside agent i's space" or $[[c]_j]_i$ "c holds in a space associated to agent j inside agent i's space" are then possible in cs. Movement of information across spaces was introduced by means of an operation $\uparrow$ called *extrusion* [5]. One can now conceive statements like $[\uparrow_i [c]_j]_i$ "agent i extrudes the information c to agent j".

The purpose of this paper is to present D-SPACES, a tool for constructing cs with space and extrusion. It is intended for simulation/verification and future software incorporation. D-SPACES was conceived as a `c++11` library, using the boost graph library[4] (`BGL`). It is thoroughly documented using `HeaderDoc`[5] and can be directly used in the OS X (XCode) and Linux (GCC+Make) environments[6]. Usage on Windows depends upon compilation of `BGL`, nonetheless the tool is sufficiently cross-platform.

The paper is divided in three sections. Section 1 provides motivation, background and basic details of the tool. Section 2 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 some operators and present the implementation of boolean algebras. Finally, Section 3 offers some concluding remarks and future endeavors of this project. Mathematical proofs and usage examples are in the Appendices.

2 Implementing Space and Extrusion in cs's

We begin by describing the class hierarchy of the tool. There are three modules implementing each constraint system, they are named `cs`, `scs` and `scse`. A fourth module, named `ba`, implements powerset boolean algebras using the functionality of all these constraint systems.

Each module parametrizes the cs elements 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`. Current instantiations support elements of type `int`, `char`, `std::string` and containers `std::vector` and `std::set` of these same types. We continue with the description of each module.

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

Definition 1. *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* c meet d*) as the maximal element* e *s.t.* $e \sqsubseteq c$ *and* $e \sqsubseteq d$ *and, (ii)* $c \sqcup d$ *(read as* c join d*) as the minimal element* e *s.t.* $c \sqsubseteq e$ *and* $d \sqsubseteq e$.

[4] `http://www.boost.org/doc/libs/release/libs/graph/`
[5] `http://developer.apple.com/opensource/tools/headerdoc.html`
[6] `http://www.lix.polytechnique.fr/~perchy/d-spaces/`

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 inverse 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*.

Definition 2. *A* constraint system *[3] is a complete lattice, that is, a lattice where the meet and join are defined for every subset of Con.*

Intuitively, a cs is an information structure, it has a global meet $\bot$ (contextually referred in cs as *true*) and a global join $\top$ (referred to as *false*). We can also define a binary implication operator $c \to d = \bigsqcap\{e \mid c \sqcup e \sqsupseteq d\}$, allowing us also to encode the negation of an element as $\neg c = c \to \mathit{false}$ [8].

Interface and usage. Table 1 describes part of the interface to the `cs` module. The input elements in `glb` and `lub` may be empty vectors, in this situation they produce the global meet and global join respectively. Similarly, in the method `add_element` the upper and lower bounds of `c` parameters may be empty, denoting $\bot$ and $\top$ respectively. Figure 2 exemplifies the usage of the `cs` module.

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	$c \to d$

Table 1. Interface to `cs`

Properties of cs. It is possible to verify properties on the structure of the cs. One important property of lattices is that of *distributivity*, defined as: for every $a, b, c \in \mathit{Con}$ $a \sqcup (b \sqcap c) = (a \sqcup b) \sqcap (a \sqcup c)$. Distributivity is necessary for *modus ponens* to hold, that is, $(c \to d) \sqcup c \sqsupseteq d$ must be true for every $c, d \in \mathit{Con}$ [5]. Distributivity can be checked with the boolean method `CS.is_distributive()`.

Spatial Constraint Systems with Extrusion. We continue by defining the remaining two constraint systems.

Definition 3. *An n-agent* spatial constraint system (scs) *is a cs equipped with n self-maps (called space functions) over its set of elements. For each map* $[\cdot]_i : \mathit{Con} \to \mathit{Con}$*: S.1* $[\mathit{true}]_i = \mathit{true}$ *and S.2* $[c \sqcup d]_i = [c]_i \sqcup [d]_i$ *for all* $c, d \in \mathit{Con}$.

We refer to S.1 as *emptiness*, intuitively signifying that no local information amounts to no information at all. S.2 is referred to as $\sqcup$-*distribution*, meaning the structure of the information is preserved inside spaces. 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 $c, d \in \mathit{Con}$. We now formalize extrusion in constraint systems.

Definition 4. *An n-agent* spatial constraint system with extrusion *is a scs equipped with n self-maps (called extrusion functions) over its set of elements. For each map* $\uparrow_i : \mathit{Con} \to \mathit{Con}$*: E.1* $[\uparrow_i c]_i = c$ *for all* $c \in \mathit{Con}$.

E.1 means that the i-th extrusion function is the right inverse of the i-th space function. One might also require (for duality reasons) that the extrusion function satisfy: E.2 $\uparrow_i(true) = true$, and E.3 $\uparrow_i(c \sqcup d) = \uparrow_i c \sqcup \uparrow_i d$ for all $c, d \in Con$.

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 and modify the space/extrusion functions as well as calculate their inverses. In Figure 3 we exemplify the usage of both modules.

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	$\uparrow_i elems = c$

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

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 is not satisfied for all $c \in Con$) [5]. Both modules offer property checking via the methods `s_properties( int i, S_FUNCTION_PROPERTY p )` and `e_properties( int i, E_FUNCTION_PROPERTY p )`. One can verify standard properties like surjectivity, $\sqcup$-distributivity (i.e. S.2, E.3) and inversion (i.e. E.1) among others.

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 if no attention is given. We discuss the most critical cases here, those of methods `leq`, `glb`, `lub` and `imp`. Recall that posets were implemented using a `BGL` graph.

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

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

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 1 is a result of Proposition 1.

Proposition 1. $max(S^l) = \underset{s_i \in S^l}{arg\,min} |{s_i}^u|$

Corollary 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 $c \to d = \bigsqcap Con = \bot$. 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. *$S' \cap S = \emptyset$ where $S = \{e \mid c \sqcup e \sqsupseteq false\}$ and $S' = (c^u \backslash \{false\})^l$.*

Application: Boolean Algebras. To provide a ready-to-use application, the module `ba` is offered as an implementation of powerset boolean algebras (ba) [6]. Given a set of atoms A, a powerset ba is a specific case of a scs-e where $Con = \mathcal{P}(A)$, $\sqcup = \cup$ (or $\cap$ if the powerset is inverted), $\sqcap = \cap$ (or $\cup$ if inverted), $\bot = \emptyset$ (or A if inverted) and $\top = A$ (or $\emptyset$ if inverted). The `ba` module can create the powerset boolean algebra by passing the set of atoms, number of agents and specifying if the powerset should be inverted. The module maps the space and extrusion functions specified through lambda functions. Because the powerset ba is also a scs-e, the module also exposes all the functionality of the constraint systems discussed here. Figure 4 is an example of using powerset ba.

3 Conclusions and Future Work

We presented a tool to simulate spatial constraint systems with extrusion called D-SPACES. We described the interface to create cs, scs and scs-e and illustrated their usage. To implement the poset of the constraint systems, we used the `BGL`'s implementation of graphs. This, together with some mathematical properties, allowed us to lower the complexity of the lattice operations, mainly when testing for correspondence and calculating meets, joins and implication of elements.

As a quick way to construct scs-e, we offered a module to create powerset boolean algebras with space and extrusion functions specified as lambda functions. As future endeavors we plan to implement more relevant constraint systems, as well as support more data types to represent elements. We would also like to see support for removing elements, as this, together with the `add_element` method, would allow to interactively manipulate and verify a scs-e.

References

1. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.Y.: Reasoning about knowledge. 4th edn. MIT press Cambridge (1995)
2. Van Ditmarsch, H., Van Eijck, J., Sietsma, F., Wang, Y.: On the logic of lying. In: Games, actions and social software. Springer (2012) 41–72
3. Saraswat, V.A., Rinard, M., Panangaden, P.: Semantic foundations of concurrent constraint programming. In: Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages. (1991) 333–352
4. Knight, S., Palamidessi, C., Panangaden, P., Valencia, F.D.: Spatial and epistemic modalities in constraint-based process calculi. In: Proceedings of the 23rd International Conference on Concurrency Theory, CONCUR 2012, Springer (2012) 317–332
5. Haar, S., Perchy, S., Rueda, C., Valencia, F.D.: 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. (2015) 161–172
6. Davey, B.A., Priestley, H.A.: Introduction to lattices and order. 2nd edn. Cambridge university press (2002)
7. Abramsky, S., Jung, A.: Domain theory. Handbook of logic in computer science (1994) 1–77
8. Vickers, S.: Topology via logic. 1st edn. Cambridge University Press (1996)

A Usage Examples

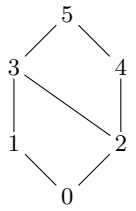

Fig. 1. A Poset

Code in Figure 2 creates a constraint system with elements $\{0, 1, 2, 3, 4, 5\}$ and the poset structure of Figure 1. Additionally, it calculates $1 \sqcup 2$, $\sqcap\{2, 3, 4\}$, $2 \to 3$ and checks if the cs is distributive.

The code of Figure 3 creates a scs out of the cs created in Figure 2 and maps some of its elements. Next, it creates a scs-e with this scs. Here, the option `EC_SUPREMA` automatically calculates the extrusion function as the join of the space function's inverse for each element. The reader is referred to [5] to see other canonical ways to calculate the extrusion function.

The last example in Figure 4 creates a powerset boolean algebra and automatically maps the extrusion function according to a user-given space function.

```
cs<int> CS( 0, 5 ); // global meet = 0, global join = 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.
```

Fig. 2. Code exemplifying the use of the `cs` module

```
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
```

Fig. 3. Code exemplifying the use of the `scs` and `scse` modules

```
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
```

Fig. 4. Code exemplifying the use of the `ba` module

B Proofs

Proposition *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 1). Furthermore, ${s_k}^u \subset {s_j}^u$ because $\sqsubseteq$ is transitive. Consequently $|{s_k}^u| < |{s_j}^u|$, a contradiction.

Proposition *2.* $S' \cap S = \emptyset$ *where* $S = \{e \mid c \sqcup e \sqsupseteq \mathit{false}\}$ *and* $S' = (c^u \backslash \{\mathit{false}\})^l$.

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