Links for Boosting Predictable Interactive Constraint Systems

Gilles Trombettoni and Bertrand Neveu

{trombe, neveu}@sophia.inria.fr
COPRIN Project, INRIA, 2004 route des lucioles
06902 Sophia.Antipolis cedex, B.P. 93, France

Abstract. Constraints are useful in interactive applications because they allow programmers to state declaratively a relation to be maintained, rather than requiring them to write procedures to maintain the relation. However, the solution predictability issue has hindered the acceptance of constraint solvers in interactive applications, such as CAD or user interfaces. Indeed, after an interaction with the system, such as the addition of a new constraint, a lot of solutions can be given whereas only a specific one is expected by the user.
This paper extends with *semantic links* the multi-way dataflow constraint model. These links specify the possible ways to solve a constraint when a given variable is modified. They are declarative and limit the number of solutions produced by local propagation. This new model can be viewed as the introduction of the event paradigm in the dataflow constraint paradigm.
We prove that this model is unfortunately NP-complete, but we identify a significant polynomial sub-class. In this class, all links are *single-method*, that is, allow at most one possible behavior. We present a new linear-time incremental algorithm which solves constraint systems with single-method links. Systems built on this model turn out to be predictable as opposite to classical multi-way constraint systems. They are more powerful than one-way constraint systems.
This new model gives promises in interactive applications, including spreadsheets and “intelligent” user interfaces.

Keywords
Constraints, Computational Complexity, User Interfaces, Spreadsheets, Event Languages

1 Introduction

Interactive applications modeled by constraint systems are always under-constrained. There exist several ways to maintain the solution after a user interaction, that is, when new constraints are incrementally added. The problem, called *solution predictability*, is to give a solution expected by the user. This issue is known to be one of the major limitations for constraint solvers in interactive applications [7, 15]. As pointed out in [15], “tools which use automatic techniques that are sometimes unpredictable have been poorly received by programmers". Our paper extends the existing multi-way constraint model with *semantic links*.

Multi-way dataflow constraints are very useful to model interactive applications in a declarative way, especially in graphical user interface (in short GUI) [17]. A multi-way constraint has several functions (called *methods*) associated to it. Executing any of these methods satisfies the constraint. The solving process of multi-way constraint problems, called *local propagation*, is run when new constraints are incrementally added. It assigns one method to each constraint before executing the selected methods in a total order. The main advantage of the multi-way constraint model is its generality. In GUIs, some constraints are not numeric, over bitmaps, colors or fonts, and others are non-linear such as in the scroll bar example shown in Section 1.1.

In the new model proposed in this paper, a set of *semantic links* is associated to every constraint c in the system in a declarative way. For each possible violation of c, a semantic link specifies which methods can be executed to maintain the consistency of c.

As an example, consider the constraint stating that two windows W_1 and W_2 must be vertically centered: $2x_1 + l_1 = 2x_2 + l_2$. If the user moves W_1 to the right, he would like to see as well W_2 move to the right to maintain the constraint (using the method $x_2 \leftarrow \frac{(2x_1+l_1-l_2)}{2}$). However, in the classical multi-way constraint model, a second solution would consist in resizing W_2, which would be a surprising behavior. In GUIs, this type of constraint is very common and underlines the implicit relations between move actions on one hand and resize actions on the other [16]. To obtain the right behavior, a semantic link from x_1 to x_2 is added to ensure that only x_2 (and not l_1 or l_2) can be modified when the centering constraint is violated by an alteration of x_1. Three other links complete the specification: from x_2 to x_1; from l_1 to l_2; from l_2 to l_1.

A motivating example is given in Section 1.1, followed by a presentation of the semantic links formalism in Section 2. We present in Section 3 a new incremental algorithm, called `AzureLink`, which solves in linear time systems with *single-method* links. Single-method links are semantic links which allow at most one method. Section 4 shows NP-completeness results which concern the model in its full generality and which highlight the interest of the polynomial sub-class restricted to single-methods links. Section 5 shows that the multi-way constraint model with single-method links appears to be a good compromise between the classical multi-way constraint model, the one-way constraint model (used in spreadsheets and also in GUIs), and event languages.

1.1 An illustrating example

The following example illustrates the known notions of constraint and method and the new notion of *link*. It also highlights the interest of the new model to embed interactive constraint applications.

Consider a scroll bar attached to a window of a text editor. The scroll bar has a thumb the height of which is proportional to the number of lines of the document that are visible in the window. The behavior and the layout of this scroll bar can be described using the following constraints:

$$\begin{cases} c_1 : \texttt{thumb.y} - \texttt{bar.y} = \texttt{bar.height} \times \frac{\texttt{document.first\_visible}}{\texttt{document.total\_lines}} \\ c_2 : \texttt{thumb.height} = \texttt{bar.height} \times \frac{\texttt{document.visible\_lines}}{\texttt{document.total\_lines}} \\ c_3 : \texttt{window.height} = \texttt{line\_height} \times \texttt{document.visible\_lines} \\ c_4 : \texttt{window.height} = \texttt{bar.height} \\ c_5 : \texttt{window.y} = \texttt{bar.y} \end{cases}$$

Fig. 1 shows the set of semantic links that define the common behavior of a scroll bar.

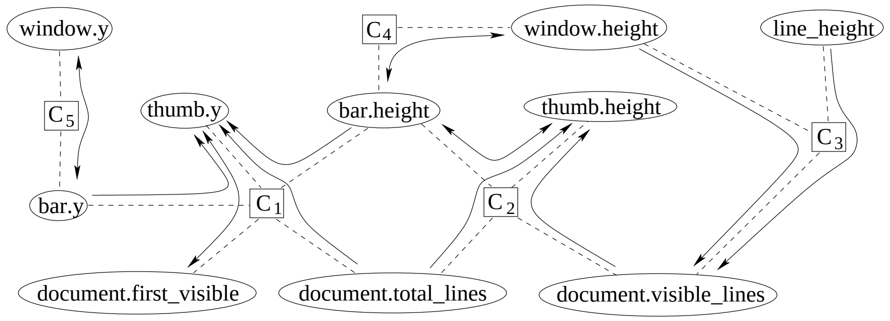

Fig. 1. Constraint graph corresponding to the scroll bar example: rectangles represent constraints; ellipses represent variables; every constraint is connected to its variables with a (dotted) edge. The arrows represent the semantic links associated to the constraints, e.g., the arrow from `bar.height` to `thumb.height` illustrates the link fired by the modification of variable `bar.height`, satisfying c_2 by calling the method: $\texttt{thumb.height} \leftarrow \texttt{bar.height} \times \frac{\texttt{document.visible\_lines}}{\texttt{document.total\_lines}}$.

For example, the *single-method link* (see definition 5) from variable `document.total_lines` to variable `thumb.height` has the following meaning: when lines are added (or removed) in the text by the user, constraint c_2 is violated; in this case, the link states that the consistency of c_2 can be recovered only by modifying the height of the thumb. This implies that it is not allowed to increase the number of visible lines in the window! However, most of the classical multi-way constraint solvers would allow the two methods to be applied indifferently, which would cause an unpredictable behavior.

This modelization can handle all possible perturbations: text edition, move and resize operations of the window or font size modifications. For instance, if the window is resized (i.e., a unary constraint is added to impose a new value to the variable `window.height`), the only *valid method graph* (see below) respecting the semantic links yields the sequence of methods which output on `bar.height`, `document.visible_lines`, `thumb.height`, `thumb.y` in order.

Remark

This example highlights that semantic links can be defined by locally looking at the constraint. When the programmer defines a new constraint in a library, he has to associate to it a set of methods and a set of semantic links. Thus, when an end-user adds this constraint in his application, methods and links are automatically added with it[1].

1.2 Other Approaches for Handling the Predictability Issue

A technique has been intensively used to improve solution predictability in constraint-based GUIs. It consists in adding unary constraints, called *stay* constraints, to "freeze" some variables. However, this approach is not declarative. Indeed, for every possible interaction with the system, the designer performs several times the same solving process: each time, he adds stay constraints manually and checks the obtained solution; he stops when the right behavior is obtained. For example, the reader can verify that the centering constraint presented in the introduction cannot be modeled with stay constraints.

To avoid this burdensome work, stay constraints can be used as soft constraints with a linear or a quadratic comparator. Soft stay constraints can be used in a local propagation solver [17, 20] or in a global solver approach based on linear programming [4, 11]. They are added on all variables and tend to produce a new solution where the sum of "errors" on every variable is minimized. However, the programmer cannot incorporate a specific semantics for a given constraint. The reader can verify that the examples in this paper cannot be made predictable with this approach.

Another approach is the constraint imperative programming framework where imperative statements and constraints can be mixed together [13].

The predictability issue appears also in CAD systems. The user gives a sketch and constraints (distance, angle) between geometric objects. When the problem is structurally well constrained, a decomposition method gives a unique solving plan. But, since the constraints are generally non linear, there exist a finite number of solutions satisfying them. In order to discriminate between these solutions, one can add, during the evaluation phase, inequality constraints or signs for angles corresponding to the sketch [7].

2 Definitions

This section first recalls basic definitions about multi-way constraint systems (see [17] or [20] for more details) and introduces then the semantic links formalism.

A *multi-way constraint* system is denoted by (V, C, M), where V is a set of variables with a current value, C is a set of multi-way constraints, and M is a set of *methods* that can satisfy the constraints individually.

[1] Interactive applications are often designed by programmers which have a deep knowledge of the model and of the underlying algorithm, as opposed to end-users which have lower technical skills and generally ignore most of this machinery.

Definition 1 *A* **multi-way constraint** *c is a relation between a set of variables $V_c \subset V$. It has one or more methods associated to it.*

A **method** *m_c^O has a set $O \subset V_c$ of* **output variables***, and a set $V_c \setminus O$ of input variables. Method m_c^O is an arbitrary piece of code which computes the value of the output variables according to the value of the input variables.*

A method with a single output variable is called **single-output** *method; a method with several output variables is called* **multi-output** *method.*

A multi-way constraint system is often represented by a constraint graph G_c. A **constraint graph** is a bipartite undirected graph whose vertices are constraints and variables. Each constraint is connected to its variables.

Local propagation [18, 2, 20] is the standard technique used to solve multi-way constraint systems, typically when new constraints, called **perturbations**, are incrementally added. It works in two phases:

- The *planning phase* assigns one method to each constraint. The result of this phase is a *valid* directed graph G_m called **method graph**.
- Two cases occur for the *evaluation phase*. When the method graph G_m contains no directed cycles, the evaluation phase executes the methods in some topological order derived from G_m. When G_m is cyclic, strongly connected components are collected and generally passed to external solvers to be satisfied as a whole.

Definition 2 *Let (V, C, M) be a multi-way constraint system, and let G_c be the corresponding contraint graph.*

A **method graph** *G_m is a directed graph obtained from the constraint graph G_c:*

- *Every constraint in G_c has one method selected for it in G_m.*
- *Every method selected in G_m implies arcs from the input variables to the constraint, and from the constraint to the output variables.*

Definition 3 *A method graph G_m is* **valid** *iff G_m has no* **variable conflicts***, that is, each variable is the output of, at most, one method.*

The problem solved by the planning phase will be called **constraint planning** in the paper. It will be called **acyclic constraint planning** if the validity condition is restricted in order to accept only acyclic method graphs. Note that this leads to two different problems the computational complexities of which are not comparable [9]. Most of the solvers, such as SkyBlue [17] and the solver from Gangnet and Rosenberg [8], handle the constraint planning problem, whereas a few solvers, such as SketchPad [18] or QuickPlan [20], tackle the acyclic constraint planning problem.

This paper proposes the extension of the multi-way constraint model with *semantic links* in order to increase the predictability of interactive constraint applications.

Definition 4 *Let $G = (V, C, M)$ be a multi-way constraint system. A* **semantic link** *l is a 3-uple (c, v, E_c^v) defined for every constraint $c \in C$ and every variable v associated to c. E_c^v is a set of methods satisfying c.*

Let m be any method which outputs on v and which satisfies a constraint $c' \neq c$. The **semantic link** *l states that if the method m is selected in a valid method graph G_m of G, then only a method $n \in E_c^v$ can be selected in G_m to solve c. We call v a "firing variable" for method n.*

Let L be the set of semantic links of G. The 4-uple (V, C, M, L) denotes a multi-way constraint system with semantic links.

A semantic link states compatibilities between any method m that outputs on v and the method n which are "causally" or "semantically" dependent: if c is violated because the value of the *firing variable* v is modified, the link decides which method can be selected to recover the satisfaction of c.

This definition implies that if variables $\{v_1, ..., v_n\}$ of c are outputs of methods in the method graph, the set of allowed methods for c is $\cap_{v_i \in \{v_1,...,v_n\}} E_c^{v_i}$.

Definition 5 *Let $l = (c, v, E_c^v)$ be a semantic link. Let M_c be the set of methods satisfying c.*
l is a **single-method link** *iff $\|E_c^v\| = 1$*
l is a **no-method link** *iff $\|E_c^v\| = 0$*
l is a **all-method link** *iff $E_c^v = M_c$*

The definitions 4 and 5 of links will facilitate the presentation of the paper. Another definition can be found below.

Definition 6 *Let $G = (V, C, M)$ be a multi-way constraint system. Defining the links amounts to associating a set of firing variables* `firing`(n) *to every method n in M. The method n can be selected only if the corresponding constraint is violated by the value modification of a variable in* `firing`(n).

It is straightforward to verify that definition 4 is equivalent to definition 6. The latter definition is probably more intuitive for programmers in practical applications: a programmer would only have to associate a set of firing variables to every defined method, implicitly adding a set of corresponding links.

3 The `AzureLink` Algorithm

We present a new algorithm, named `AzureLink`, to solve the *constraint planning with semantic links* problem, that is, to find a valid method graph respecting semantic links. As input, the constraint graph can include both single-output methods and multi-output methods, but only no-method links and single-method semantic links can be defined. This restriction allows one to perform a backtrack free process to recover the consistency. Indeed, when a method m is selected for a given constraint c, the links force the selection of exactly one method for every constraint connected to an output variable of m.

When a constraint is incrementally added to the system, `AzureLink` produces a unique valid method graph, and does it in linear time. Thus, the predictability issue can be tackled in a very efficient way. Since Section 4 will prove that the constraint planning problem with semantic links is NP-complete in general, the good properties of AzureLink highlight the interest of the sub-class restricted to only single-method links and no-method links.

3.1 Description of `AzureLink`

Algorithm 1 details `AzureLink` which is called with a set of constraints and a set of single-method links and no-method links associated to the constraints. If `AzureLink` succeeds, it returns *true* and G is a valid method graph; it returns *false* when it fails. The work of `AzureLink` on the scroll-bar example is traced on Fig. 2 at the end of the section.

```
algorithm AzureLink (C: set of constraints; L: set of links, in-out G: method
graph): boolean
    while C ≠ ∅ do
        let c ∈ C and let M_c be the methods satisfying c
        fired? ← false
        while (M_c ≠ ∅) and (not fired?) do
            let m ∈ M_c
            M_c ← M_c \ {m}
            fired?←Enforce(m, G, L)
        end
        if (not fired?) then exit and return (false)
        C ← C \ {c}
    end
    return (true) /* solution found */
end.
```

Algorithm 1: The `AzureLink` algorithm

`AzureLink` is based on the *propagation of conflicts* scheme used in most of the local propagation solvers such as Thinglab or SkyBlue [17].

Algorithm 1 succeeds if all the constraints in C are successfully added to the method graph G (outer loop). In this case, it is ensured that one method is selected for every constraint in the method graph G at the end.

If one method m is successfully selected for a constraint c in C (the first one), c and m are added to the current method graph, and the process is reitered on another constraint. The handling of a constraint c fails (and Algorithm 1 as well) if no method can be enforced for c (inner loop).

Note that `AzureLink` is generally run with the following parameters:

– *From scratch*: G is empty and C contains all the constraints of the problem.

– *Incremental version*: G is not empty and C contains only one constraint to be added. (In this case, the outer loop is useless.)

The key point of the `AzureLink` algorithm is the `Enforce` function. This function imposes the selection of the method m (m_{init} inside the function) in G and checks whether the problem made of the constraint satisfied by m_{init} and the constraints in G is consistent.

function `Enforce` (m_{init}*: a method, in-out* G*: method graph,* L*: set of links): boolean*

  $M \leftarrow \{m_{init}\}$; `possible?` $\leftarrow$ *true* ; $M_{new} \leftarrow \emptyset$

  while (`possible?`) **and** ($M \neq \emptyset$) **do**

    let $m \in M$ and **let** c be the constraint satisfied by m

    if ($m = \perp$) **or** (`VariableConflict`(m, M_{new})) **or** ($\exists$ *method* $m' \in M_{new}$ *satisfying* c ($m \neq m'$)) **then**

      `possible?` $\leftarrow$ *false*

    else

      $M \leftarrow M \setminus \{m\}$

      if $m \notin M_{new}$ /* *propagation process* */ **then**

        $M_{new} \leftarrow M_{new} \cup \{m\}$

        let N be the set of methods m_i satisfying c_i such that:

        – v_i is an output variable of m

        – c_i is a constraint in G connected to v_i

        – $(v_i, c_i, \{m_i\}) \in L$

        $M \leftarrow M \cup N$

    end

  end

  if `possible?` [**and** `Acyclic?`(M_{new}, G)] **then**

    foreach $m \in M_{new}$ **do** `Update` (G, m)

    return *true*

  else

    return *false*

  end

end.

The function `Enforce` performs a propagation process in the current method graph G. The function manages a set M of methods to be handled, and a set M_{new} of methods that will be selected in the method graph G if the propagation process succeeds. (The side-effects in the method graph occur in the `for` loop at the end of the function.) At the beginning, M contains only the method m_{init}, i.e., the method which `AzureLink` tries to enforce. A method m is transferred from M to M_{new} when m is consistent with m_{init}, according to variable conflicts and semantic links. Such a selection may cause propagation in the neighbor constraints c_i: methods m_i satisfying c_i are pushed into M when an output variable v_i of m is connected to a constraint c_i in G (the execution of m may violate c_i) and when a link allows the selection of m_i. Since the links are single-output or no-output, at most one method m_i satisfying c_i can be chosen.

If no failure occurs, the process is reitered until M becomes empty. In this case, every method m in M_{new} is added to the method graph G by the `Update` function. If the constraint c satisfying m has another method n currently selected for c in G, the current selection is changed from n to m. In the case where only an acyclic method graph is sought for, a test of acyclicity is added in G. This test is performed by the function `Acyclic?` on the sub-graph of G including methods in M_{new}.

The `Enforce` function fails when a method cannot be transferred from M to M_{new} (`possible?`=*false*). There are three causes of failures:

- *No-method links*: A method $m_i = \perp$ has been added to M if the link was no-method. This means that no method can be selected for constraint c_i when v_i is modified.
- *Variable conflicts*: The process may also fail when a variable conflict occurs between m and another method in M_{new}, both methods outputing on a same variable.
- *Contradictory links*: It is possible that one wants to add a method m to M_{new}, whereas another method m', satisfying the same constraint, has previously been added to M_{new}. This means that the two single-output links that pushed m and m' on M are contradictory.

It should be noticed that the propagation process performed by the `Enforce` function is backtrack-free thanks to the links that can be only single-method or no-method. `AzureLink` can change several times the method selected for a given constraint, but at most once at each call of the `Enforce` function.

More generally, it should be also noticed that this propagation of conflicts scheme performed in `AzureLink` has certain advantages as compared to reactive algorithms embedded in event languages (see Section 5.3). In particular, it is useful to detect contradictory links or to not propagate further when a method has already been pushed in M_{new}. These tasks cannot be performed by a reactive algorithm[2]. They are specific to planning phase algorithms and avoid executing a method several times in the execution phase.

Figure 2 shows a valid method graph that is built by `AzureLink` on the scroll bar example. We assume that initially the method graph is empty and C contains all the constraints in the problem, and that the constraints in C are added in lexicographical order.

3.2 Incremental version of the `Enforce` function

The `Enforce` function is incremental in the sense that it does not necessarily traverse all the constraints in the current graph G at each time. In the example

[2] i.e., no planning phase is performed, and a method is executed as soon as it is selected...

of Fig. 2 (D), when the method of constraint c_4 is enforced, the constraint c_3 is not considered by the function because no output variable of a method in M_{new} is connected to c_3.

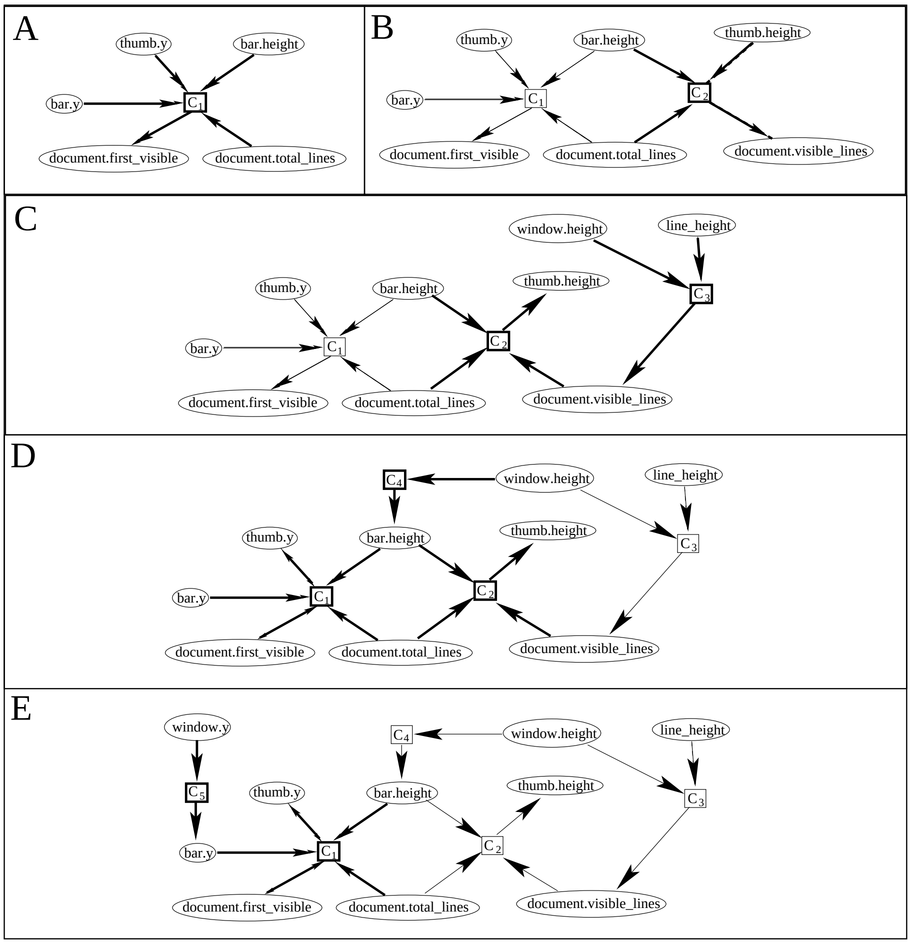

Fig. 2. `AzureLink` on the scroll-bar example. The different frames show a valid method graph after every call to the `Enforce` function. Methods in bold are those pushed into M_{new} by the latter call to function.

It is possible to render the `Enforce` function even more incremental by taking into account the methods selected in previous calls to the `Enforce` function. Line 11 of the function must be modified as follows: **if** $m \notin M_{new}$ **and** m is not selected for c in G **then**...[3]

[3] The acyclicity test, if performed, must also be modified because the set M'_{new} obtained by the incremental version is a subset of the set M_{new} obtained by the presented version.

This type of improvement is implemented in most of the existing planning phases, including DeltaBlue [6], SkyBlue [17] and QuickPlan [20]. In the same example (see Fig. 2 (D)), the `Enforce` function selects the method which outputs on `thumb.height` whereas this method has already been selected in the graph G. Thus, checking current method selections in G allows the algorithm to avoid useless propagations: e.g., the constraints c_2 in Fig. 2 (D) and c_1 in Fig. 2 (E) would not be considered by the incremental version of `AzureLink`.

3.3 Properties of the `Azurelink` algorithm

Proposition 1 *The set M_{new} of methods which can be selected by the* `Enforce` *function is unique.*

The proof of Proposition 1 is straightforward. It comes from the fact that single-method and no-method links render deterministic the selection of methods. This has a very positive impact on the predictability issue in incremental applications. Indeed, when the perturbation consists of one constraint associated to only one possible method (e.g., modifying the value of a variable or the value of a cell in a spreadsheet), then there exists only one valid method graph to be evaluated in order to take the perturbation into account.

Lemma 1 *The* `Enforce` *function is correct, complete, and is time $O(n)$. It is time $O(k" \times n)$ when the acyclicity test is performed. n is the number of constraints in the method graph G, and $k"$ is the maximum arity of the constraints.*

Proof

Starting from the initial method m_{init} to be enforced, the function handles a constraint in G at most once and is therefore $O(n)$. Performing the acyclicity test is $O(k" \times n)$ (this amounts in a topological sort), which dominates the overall complexity.

It is correct since the propagation process ensures that every constraint which needs to be handled is handled by the function, and that no variable conflict can occur thanks to the `VariableConflict` function. Also, all the involved semantic links are checked.

Since no choice points are made during the construction of M_{new} and since the failures can only occur during that construction, the algorithm is complete. □

Proposition 2 *Let n be the number of constraints in C (G is initially empty), let k' be the maximum number of methods per constraint, and let $k"$ be the maximum arity of the constraints.*

`AzureLink` *is correct, complete and is time $O(k' \times n^2)$. It is time $O(k' \times k" \times n^2)$ when acyclic solutions are sought for.*

Proof

The complete proof can be made by induction over the set of constraints handled.

For every handled constraint c, all the possible method selections are tried. Furthermore, for each choice of method, Lemma 1 ensures the correctness and the completeness of the process. Therefore `AzureLink` is correct and complete.

`AzureLink` performs at most $k' \times n$ calls to the `Enforce` function which is itself $O(n)$, or $O(k" \times n)$ if only acyclic method graphs are sought for. □

The presentation of `AzureLink` and of its properties is now followed by the study of the computational complexity of the semantic links formalism in its full generality.

4 Computational Complexity of the constraint planning problem with semantic links

This section studies the computational complexity of the constraint planning problem where instances contain semantic links.

It is known that (cyclic) constraint planning is NP-complete [19]. However, when the instances contain only single-output methods, it can be solved in polynomial time by the maximum-matching algorithm [8]. It is also known that the acyclic constraint planning problem is polynomial since it can be solved in polynomial time by the PDOF algorithm [20]. The following two theorems show that constraint planning and acyclic constraint planning become NP-complete when semantic links are added to the model.

Theorem 1 *Let R be the class of multi-way constraint systems with single-output methods only, and semantic links.*
Then proving the existence of a valid method graph (cyclic or not) for any instance in R is NP-complete.

`lcp` stands for the "Constraint Planning with semantic Links" problem in the following. The sub-class of `lcp` where instances contain only single-output methods is called `lsocp`. Theorem 1 states that `lsocp` is NP-complete, so that `lcp` is also NP-complete.

Annex A proves that `2-output constraint planning` [19] can be polynomially reduced into `lsocp`.

Theorem 2 *Let R be the class of multi-way constraint systems with single-output methods only, and semantic links.*
Then proving the existence of a valid `acyclic` *method graph for any instance in R is NP-complete.*

`lacp` stands for the "Acyclic Constraint Planning with semantic Links" problem in the following. The sub-class of `lacp` where instances contain only single-output methods is called `lsoacp`. Theorem 2 states that `lsoacp` is NP-complete, so that `lacp` is also NP-complete.

Annex B proves that 3-SAT [9] can be polynomially reduced into `lsoacp`.

Theorems 1 and 2 show that the constraint planning problem becomes NP-complete when semantic links are added to the model. The result holds even when only single-output methods are defined, and when cyclic method graphs are allowed (`lcp`) or when they are not (`lacp`).

These results show that it is unlikely to translate "easily" semantic links in the classical multi-way constraint model. Especially, modifying the set of defined methods or using constraint hierarchies with locally-better criteria [17] are not promising approaches to embed semantic links (if $P \neq NP$).

Table 1 sums up the computational complexities obtained and highlights four polynomial sub-classes of `lacp` and `lcp` which are useful in practice[4].

	General	**All-method**	**Single-method**
`lcp`	NP-complete	P (`Max.-matching`) or NP-complete*	P (`AzureLink`)
`lacp`	NP-complete	P (`PDOF`)	P (`AzureLink`)

Table 1. Computational complexity of the constraint planning problem with semantic links. The different complexities of the two problems (cyclic method graphs allowed or not) are given according restrictions on the allowed semantic links. The column "All-method" refers to classical multi-way constraints systems for which constraint planning is a sub-class of `lcp` (or `lacp`). Indeed, a multi-way constraint can be seen as a constraint for which the links are all-method. The problem is NP-complete ($*$) if multi-output methods are allowed. The column "All-method" refers to the sub-class restricted to single and no-method links.

The constraint planning problem becomes easy when instances contain only all-methods links, or when they contain only single-method and no-method links. A natural question is therefore whether the problem comes down to P if instances mix no-method, single-method and all-method links (but no other types of links).

Corollary 1 *Constraint planning with only no-method, single-method and all-method links is NP-complete.*

Acyclic constraint planning with only no-method, single-method and all-method links is NP-complete.

Proof
This result comes directly from the reductions involved in theorems 1 and 2. The

[4] Note that `lacp` and `lcp` become also trivially polynomial when applied to trees (i.e., acyclic constraint graphs), but a few practical instances are expected in these sub-classes.

instances of `lcp` and `lacp` obtained by these reductions are exactly in the class analyzed in Corollary 1.[5] □

The technical presentation of this new model is now over and is followed by a comparison with existing models.

5 Comparison With Related Models

We hope that the semantic links formalism will have a significant impact in interactive applications and this section collects advantages over the classical multi-way constraint model, over the spreadsheet model, and over event languages.

5.1 Comparison with the multi-way constraint model

It is interesting to note that a classical multi-way constraint system accepts more solutions than the same system where the methods are restrained by single-method or no-method links. However, when all links can be "naturally" expressed with at most one possible method, the additional solutions of (pure) multi-way constraints are not relevant. The reader can verify this on the scroll-bar example. On one hand, it is true that all-method links are often not useful in applications, but on the other hand, experiments on real interactive applications based on the single-method links model will be necessary to verify whether single-method and no-method links are sufficient in practice.

Although numerous multi-way constraint solvers have been designed in the past, such as SketchPad [18], Thinglab [2], DeltaBlue [6], SkyBlue [17], and QuickPlan [20], the technique has not been adopted in commercial GUI builders [15]. The lack of predictability of these tools is one major explanation of this relative failure. Thus, embedding `AzureLink` in a constraint-based GUI builder, such as Multi-Garnet [17], should give to a designer a declarative, powerful and predictable tool, and should increase the acceptance of the model.

Note that it is straightforward to extend `AzureLink` in order to tackle *constraint hierarchies* with a locally-better criterion [17]. This requires plunge `AzureLink` in two outer loops, as done in SkyBlue and QuickPlan[6]. See [20] for more details.

5.2 Comparison with the spreadsheet model

The simplicity and the predictability of the spreadsheet model explains the success of one of the most popular software in the world.

[5] "Mixing" two polynomial sub-classes does not necessarily yield another polynomial sub-class, e.g., the class of instances made of either clauses with 2 literals (2SAT), or Horn clauses.

[6] (For a specialist in local propagation !) To add a constraint: until a fix-point is reached, one loop enforces constraints in decreasing order of priority; the other removes constraints in increasing order of priority...

The spreadsheet model is also known as the one-way constraint model since the "constraint" system is made of constraints which accept only one method (i.e., the formula). The spreadsheet model is also the base for several constraint GUI builders, such as Penguins [12], Garnet and Amulet [14].

Note that the spreadsheet model is a sub-class of the multi-way constraint model, of the semantic links constraint model, and of the single-method links constraint model.

The single-method links model is more attractive than the spreadsheet model for several reasons:

- As opposed to the spreadsheet model, the single-method links model allows one to define true (multi-directional) relations, which improves declarativity and the number of possible interactions.
- Both models define applications with nearly the same degree of predictability. More precisely, a one-way constraint system defines exactly one method graph, whereas at most k' method graphs could be built by `AzureLink` when a perturbation with k' possible methods is added. As pointed out in Section 3.3, only one method graph to be executed exists when only one variable value is modified.
- When one constraint is incrementally added, `AzureLink` is linear in the number of constraints and thus has the same asymptotic complexity as a one-way constraint solver (if the maximum number of methods per constraint k' is constant). Remind that one-way constraint solvers only have to perform the evaluation phase, that is, the topological sort of the method graph.

5.3 Comparison with event languages

Event languages [10] are simple to be used by non-specialists and produce predictable interactive applications. This explains an increasing success of the paradigm and the raise of commercial tools using event languages concepts, such as Visual Basic, HyperTalk or Java Beans.

With an event language, the user must define rules that contain the code to be executed (i.e., the handler) when a causal event occurs – such as manipulation of an input device. In the first example about the centering constraint, modifying the horizontal position of window W_1 could be considered an event firing the handler that modifies the horizontal position of window W_2. (That handler can also broadcast the event that W_2 has changed...) More generally, a handler is an arbitrary piece of code. It may fire other events or can produce side-effects in certain systems. The corresponding solvers generally use an ad-hoc algorithm, or are based on a lower level kernel [5]. However, in all cases, solvers are based on a reactive (event-driven) algorithm that fires events as soon as a rule allows it.

The semantic links introduced in this paper are in a sense a specific type of such rules. With this restrictive way to define events in the semantic links formalism, come some good properties:

- The semantics is formally defined.

– Side-effects are avoided. (Only variables in the constraint system can be modified.)
– The computational complexity has been well studied: NP-completeness results give the limits of the approach in interactive applications, and AzureLink is an incremental algorithm than can tackle the interesting polynomial sub-class.
  No formal computational complexity studies have been previously carried out on event languages, due to the large amount of different problems tackled by different systems. Furthermore, trying to characterize problems for which side-effects are allowed has no sense.
– As opposed to reactive algorithms, the planning phase performed by AzureLink has several advantages. It avoids constraints to be satisfied several times and ensures termination (no infinite loop can be done).

5.4 Summary

The semantic links formalism can be viewed as plunging the event paradigm in the dataflow constraint formalism. The relevance of this new formalism amounts in answering to the following question: is the single-method links formalism sufficient to deal with interactive applications in practice? An intuition for a positive answer is based on the success of a simpler formalism, that is, one-way constraints, in spreadsheets and GUIs. Real experiments are necessary to give an answer.

6 Conclusion and Future Works

The paper has introduced the semantic links constraint formalism to deal with the solution predictability issue in interactive applications. The corresponding constraint planning problem has been proven to be NP-complete. An interesting polynomial sub-class has been found, where only single-method and no-method links can be defined. Based on the propagation of conflicts scheme, a new algorithm, called `AzureLink`, has been presented to tackle this polynomial sub-class. When a new constraint is incrementally added to the system, `AzureLink` is linear in the number of constraints and produces exactly one plan (method graph) to be evaluated.

The semantic links constraint formalism appears to be more powerful than the spreadsheet model and has thus the potential to lead to new generations of spreadsheets and constraint-based user interfaces.

The notion of links appears also in event languages. The semantic links formalism can be viewed as plunging the event paradigm in the dataflow constraint formalism. This restrictive way to define events offers formal computational complexity studies and a solver with good properties such as a linear time complexity and termination.

Other application fields should be investigated, especially those where multiway or one-way constraint models have already been applied, such as 3D animation [1].

Other techniques could enlarge the power of `AzureLink`. Constraint hierarchies can be taken into account using a slight improvement of the algorithm. Since numerous constraints in GUIs are linear, a cooperative architecture, such as Ultraviolet [3], could make `AzureLink` collaborate with an incremental simplex algorithm.

References

1. Jean-Francis Balaguer and Enrico Gobbetti. 3d user interfaces for general-purpose 3d animation. *Computer*, 29(8):71–78, 1996.
2. Alan Borning. THINGLAB*: A Constraint–Oriented Simulation Laboratory*. PhD thesis, Stanford University, 1979.
3. Alan Borning and Bjorn N. Freeman-Benson. Ultraviolet: A constraint satisfaction algorithm for interactive graphics. *Constraints*, 3(1):9–32, 1998.
4. Alan Borning, Kim Marriott, Peter Stuckey, and Yi Xiao. Solving linear arithmetic constraints for user interface applications. In *Proceedings of the 1997 ACM Symposium on User Interface Software and Technology*, pages 87–96, 1997.
5. Frédéric Boussinot and Jean-Ferdinand Susuni. The sugarcubes tool box: A reactive java framework. *Software Practice and Experience*, 28(14):1531–1550, December 1998.
6. Bjorn Freeman-Benson, John Maloney, and Alan Borning. An incremental constraint solver. *Communications of the ACM*, 33(1):54–63, January 1990.
7. Ioannis Fudos and Christoph Hoffmann. A graph-constructive approach to solving systems of geometric constraints. *ACM Transactions on Graphics*, 16(2):179–216, 1997.
8. Michel Gangnet and Burton Rosenberg. Constraint programming and graph algorithms. In *Second International Symposium on Artificial Intelligence and Mathematics*, January 1992.
9. M. Garey and D. Johnson. *Computers and Intractibility*. Freeman and Co, 1979.
10. Ralph D. Hill. Supporting concurrency, communication, and synchronization in human-computer interaction – the Sassafras UIMS. *ACM Transactions on Graphics*, 5(3):179–210, July 1986.
11. Hiroshi Hosobe. A scalable linear constraint solver for user interface construction. In *Constraint Programming – CP'2000*, pages 218–232, 2000.
12. Scott Hudson. User interface specification using an enhanced spreadsheet model. *ACM Transactions on Graphics*, 13(3):209–239, July 1994.
13. Gus Lopez, Bjorn Freeman-Benson, and Alan Borning. Implementing constraint imperative programming languages: the Kaleidoscope'93 virtual machine. In *OOPSLA'94*, October 1994.
14. B.A. Myers, R.McDaniel, R.Miller, A. Ferrency, A. Faulring, E. Borison, B. Kyle, A. Mickish, A. Klimovitski, and P. Doane. The amulet environment: New models for effective user interface software development. *IEEE Transactions on Software Engineering*, 23(6):347–365, 1997.
15. Brad Myers, Scott E. Hudson, and Randy Pausch. Past, present and future of user interface software tools. *ACM Transactions on Computer-Human-Interaction*, 7(1):3–28, 2000.
16. Bradley Myers. *Creating User Interfaces by Demonstration*. Academic Press, 1988.
17. Michael Sannella. *Constraint Satisfaction and Debugging for Interactive User Interfaces*. PhD thesis, Department of Computer Science and Engineering, University of Washington, Seattle, 1994.

18. Ivan Sutherland. *Sketchpad: A Man-Machine Graphical Communication System.* PhD thesis, Department of Electrical Engineering, MIT, 1963.
19. Gilles Trombettoni and Bertrand Neveu. Computational complexity of multi-way, dataflow constraint problems. In *International Joint Conference on Artificial Intelligence, IJCAI'97*, pages 358–363, 1997.
20. Bradley Vander Zanden. An incremental algorithm for satisfying hierarchies of multi-way, dataflow constraints. *ACM Transactions on Programming Languages and Systems*, 18(1):30–72, 1996.

A Proof of Theorem 1

Theorem 1 *Let R be the class of multi-way constraint systems with single-output methods only, and semantic links.*
Then proving the existence of a valid method graph (cyclic or not) for any instance in R is NP-complete.

We prove that `2-output constraint planning` [19] can be polynomially reduced into `lsocp`.

Polynomial reduction

`2-output constraint planning` is NP-complete (see [19], theorem 2). Instances of this problem are made of multi-way constraints with single-output methods and of "flip" constraints with 2-output methods: flip constraints have four variables x_1, x_2, y_1, y_2 and two 2-output methods (one that outputs on x_1 and x_2, the other that outputs on y_1 and y_2).

The polynomial transformation from `2-output constraint planning` to `lsocp` is based on a "local replacement" [9] of flip constraints: every flip constraint is replaced by a specific pattern, called "flip" pattern, made of single-output constraints and semantic links, as shown in Fig. 3. Constraints with only single-output methods are left unchanged.

Proof

We will now prove that the reduction above is polynomial, that `lsocp` is in NP, and the equivalence between the existence of (i) a solution S to a 2-output constraint planning instance and (ii) a solution S' to `lsocp`.

The reduction is polynomial since a constant number of constraints, variables, methods and semantic links are generated for every flip constraint. `lsocp` is also trivially in NP since verifying the validity of semantic links in a method graph is constant for every pair made of one variable and one constraint.

The equivalence between solutions S and S' is based on the following lemma.

Lemma 2 *Consider an instance I' of* `lsocp` *obtained by the polynomial transformation. Let S' be a solution of I'. Let P be any flip pattern in I'.*
If the variable y'_1 or y'_2 (resp. x'_1 or x'_2) of P is output in S' by a method not in P, then variables x', x'_1 and x'_2 (resp. y', y'_1 or y'_2) of P are output by methods in P.

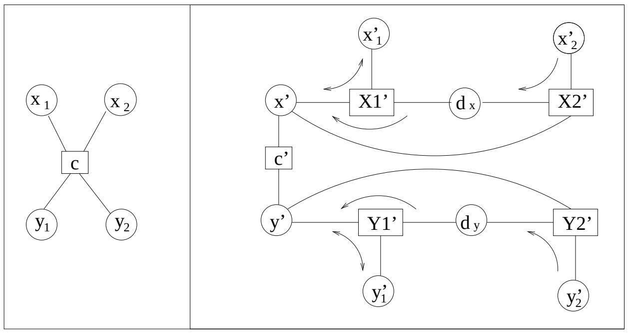

Fig. 3. Polynomial reduction for `lsocp`. Every "flip" constraint of a 2-output constraint planning instance (left) is polynomially transformed into a "flip" sub-graph in a `lsocp` instance (right). All links which are not specified on the figure correspond to all-method ones.

The proof is straightforward since no other choice is possible for building a valid method graph. Based on this lemma, we obtain:

(i) → **(ii)** If method $m_c^{x_1,x_2}$ is selected in S, one selects methods $m_{c'}^{x'}$, $m_{X1'}^{x'_1}$, $m_{X2'}^{x'_2}$, $m_{Y1'}^{y'}$, $m_{Y2'}^{d_y}$ in S', which does not cause variable conflicts and does not violate semantic links.

(ii) → **(i)** If the method $m_{c'}^{x'}$ is selected in S', one selects the method $m_c^{x_1,x_2}$ in S.

(The same method selections are made in S and S' for the constraints with only single-output methods.)

B Proof of Theorem 2

Theorem 2 *Let R be the class of multi-way constraint systems with single-methods only, and semantic links.*
Then proving the existence of a valid `acyclic` *method graph for any instance in R is NP-complete.*

We prove that 3-SAT [9] can be polynomially reduced into `lsoacp`.

Polynomial reduction

Let $F = (L, C)$ be an instance of 3-SAT, where L is the set of boolean variables in F and C is the set of clauses in F. Let $G' = (V', C', M', L')$ be the instance of `lacp` obtained by a polynomial reduction applied to F, as follows:

– Variables in V' are divided into three sets $VarL$, $VarC$ and $VarExitC$. Constraints in C' are divided into three sets $ConstL$, $ConstC$ and $ConstExitC$.

– Every clause c of C is a disjunction of 3 literals l_1, l_2 and l_3. The clause c corresponds to one constraint *c-at-least* of $ConstC$. The constraint *c-at-least* is linked to three variables *c-*l_i*-pos* or *c-*l_i*-neg* (whether l_i is positive or negative) of $VarC$ output by three single-output methods ($i \in [1..3]$).
– Every boolean variable l in L corresponds to one binary constraint l' in $ConstL$, two variables in $VarL$ and two methods. The two variables are called "true" and "false" variables of l'.
– Every literal l_i ($i \in [1..3]$) in every clause c of C corresponds to one ternary constraint *c-*l_i*-exit* of $ConstExitC$ and three methods. The constraint *c-*l_i*-exit* is linked to the variable *c-*l_i*-pos* (or *c-*l_i*-neg*) and to a variable called "exit" variable of *c-*l_i*-exit*. It is also connected to the "false" variable of constraint l'_i if the literal l_i is positive in c, or to the "true" variable of constraint l'_i if l_i is negative.
Two semantic links are associated to the constraint *c-*l_i*-exit*: *(c-*l_i*-exit, c-*l_i*-pos,* $\{m^{exit}_{c\text{-}li\text{-}exit}\}$*)*[7] and
(c-l_i*-exit, false,* $\{m^{c\text{-}li\text{-}pos}_{c\text{-}li\text{-}exit}\}$*)*[8].
Every other pair (v, c) ($v \in V'$, $c \in C'$) has an all-method link (see definition 5) which is not represented on Fig. 4.

Fig. 4 shows an instance of `lsoacp` obtained by reducing the instance *(*$\{a,b,c\}$*,* $\{1{:}(\neg a \vee \neg b \vee c)\ 2{:}(a \vee \neg b \vee \neg c)\}$) of 3-SAT.

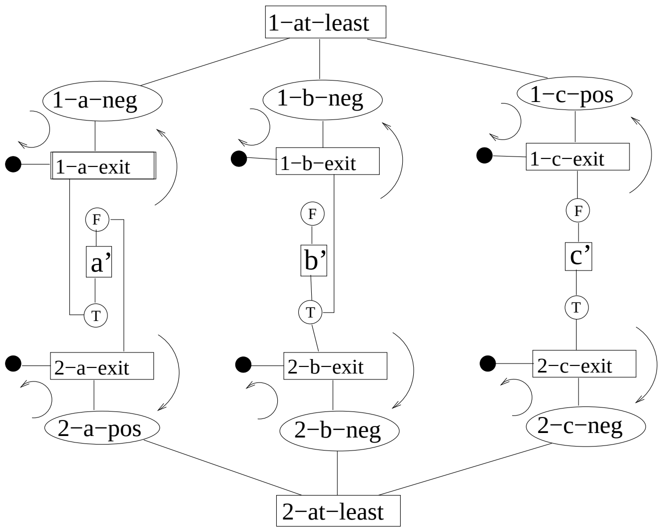

Fig. 4. An instance of `lsoacp` obtained by a reduction of a 3-SAT instance. Black circles represent "exit" variables. All links which are not specified on the figure correspond to all-method ones.

---
[7] or *(c-*l_i*-exit, c-*l_i*-neg,* $\{m^{exit}_{c\text{-}li\text{-}exit}\}$*)*
[8] or *(c-*l_i*-exit, true,* $\{m^{c\text{-}li\text{-}neg}_{c\text{-}li\text{-}exit}\}$*)*

Proof

As for the proof of theorem 1, it is straightforward to check that the reduction is polynomial and that `lacp` is in NP. Checking the acyclicity of a method graph amounts to a linear time topological sort.

We prove the equivalence between the existence of (i) a solution S to any instance F of 3-SAT and (ii) a solution S' to the instance G' of `lacp` obtained after having applied on F the reduction above.

(i) $\rightarrow$ **(ii)** Consider the value of every boolean variable l in S.
If l is true, the method $m_{l'}^{true}$ is selected in S'. A method $m_{\text{c-l-exit}}^{\text{c-l-neg}}$ is selected for every constraint *c-l-exit* linked to the "true" variable.
If l is false, the method $m_{l'}^{false}$ is selected in S' as well as methods $m_{\text{c-l-exit}}^{\text{c-l-pos}}$.

Consider now every clause c in S. Choose one literal l of c that satisfies c in S and then select method $m_{\text{c-at-least}}^{\text{c-l-pos}}$ (or $m_{\text{c-at-least}}^{\text{c-l-neg}}$) in S'. For every constraint in $ConstExitC$ which has no method yet selected, select the method determining its "exit" variable.

The validity of S' comes from the conjunction of the following four points:

- By construction, every constraint has a method selected in S'.
- By construction, no variable conflicts occur.
- By construction, the semantic links are respected.
- Moreover, no cycles appear in the method graph thanks to the "exit" variables and the semantic links that have been especially generated for this reason. Indeed, any cycle would necessarily pass through a constraint *c-at-least* of $ConstC$. However, the variable l determined by the selected method is connected to an "exit" constraint and the semantic links impose the selection of the method determining the "exit" variable.

(ii) $\rightarrow$ **(i)** Consider the method selected for every constraint *c-at-least* of $ConstC$ in S'. If the literal l corresponding to the variable determined by the method is negative, then instantiate l with the value *false*, otherwise select the value *true*. Instantiate the other variables indifferently with *true* or *false*. We can prove that the valuation then obtained is a solution S of F.

First, by construction, each clause is satisfied by the chosen valuation of the literals.

Second, it must be ensured that a same boolean variable is not instantiated with both *false* and *true* in different clauses. Suppose that this situation could occur for a boolean variable l which appears as a positive literal in a clause c and as a negative literal in a clause c'. This would mean that the two methods $m_{\text{c-l-exit}}^{exit}$ and $m_{\text{c'-l-exit}}^{exit}$ would have been selected in S'. In this case, no method could have been selected for the constraint l' of $ConstL$, which leads to a contradiction. Indeed:

- If the method determining the "true" variable were selected, then a variable conflict would have occured on variable *c'-l-neg*.
- If the other method were selected, a variable conflict would have occured on variable *c-l-pos*.

□