Spatial Reasoning about String Loops and Holes in Temporal ASP

Pedro Cabalar[1] , **Paulo E. Santos**[2,3]
[1]University of A Coruña, Spain
[2]Flinders University, Adelaide, Australia
[3]FEI, São Paulo, Brazil
cabalar@udc.es, paulo.santos@flinders.edu.au

Abstract

This paper introduces a new formalism for the automated solution of spatial scenarios involving strings and holed objects. In particular, we revisit a previous formalisation that allows string loops to be treated as holes, but make a substantial modification by removing a previous limitation that prevented a string to cross its own loops. The formalisation introduced in the present paper relies on string segments as basic entities and achieves a greater degree of elaboration tolerance by using inertia to describe those parts of the physical scenario that are unaffected by a given action. As a representation language, we have used Temporal Answer Set Programming since it provides a simple and natural way to deal with time and inertia while, at the same time, it is accompanied by the automated tool `telingo` that allows a systematic testing of the effects of any sequence of actions. As an illustrative example, we have studied the African Ring puzzle, a problem involving loops crossed by a unique string, and provided the first formalisation of its solution, to the best of our knowledge.

1 Introduction

One of the long-term goals of Artificial Intelligence (AI) is to endow computers with *commonsense reasoning*, a topic that dates back to the very beginning (McCarthy, 1959) of the area of Knowledge Representation (KR). A device with common sense should be capable of making similar assumptions as those made by humans in ordinary situations of their daily life. These assumptions involve common knowledge about their physical environment, the behaviour of other agents, the existing interactions and their possible effects, all of them, crucial capabilities for the development of intelligent robotic systems. As a challenging example, think for instance how humans can easily manipulate flexible objects like strings and make them interact with holed objects. People soon learn to deal with ordinary situations like putting a belt on a pair of trousers or tying a shoelace, but may also be trained to handle more complex scenarios involving rope handling as in sailing, suturing, knitting or climbing. If we want to emulate this behaviour on a computer or a robot, a first question we must answer is *how do we formally represent strings?* It seems clear that persons rarely handle accurate measurements or three-dimensional curves but, instead, they normally reason, learn and talk about strings using *qualitative* terms. One possibility explored by some authors has been to use *Knot Theory* (Menasco, 2005; Kauffman, 2005) in robotics as, for instance, in the robotic systems for autonomous knot tying tasks (Sanchez et al., 2018). Actions on flexible objects in this context were defined as an extension of the Reidemeister (1983) moves. This representation is suitable for the identification of string states from a computer vision system; however, it falls short in the context of problem solving, which is the main purpose of the present work. In fact, in a natural language description of a string-manipulation problem we normally find concepts closer to the possible actions and constraints imposed by the problem, rather than fine-grained, planar topological information as the one used in knot theory.

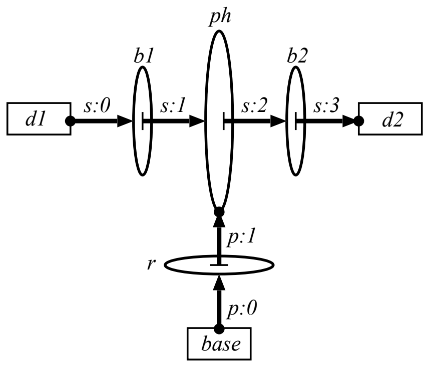
In a series of papers, (Cabalar and Santos, 2006, 2011, 2016) plus (Santos and Cabalar, 2008, 2013b, 2016) an incremental formalisation was developed of scenarios involving strings and holed objects. Following a bottom-up methodology well-established in KR, we have considered different families of puzzles involving strings, starting from simpler cases and gradually increasing the complexity of the operations required for solving the puzzle. In a first stage, Cabalar and Santos (2006) formalised a family of puzzles that can be solved without forming loops in the strings. Later on, this restriction was removed in (Cabalar and Santos, 2011; Santos and Cabalar, 2016) so that, adding new actions for *picking* and *pulling* from string segments, loops could be dynamically created and destroyed. More importantly, string loops also behaved as holes and could be, in their turn, crossed by other strings. However, an important limitation of this last formalisation was the *impossibility of handling strings that crossed their own loops*. This impossibility did not come from the string state representation itself, which actually allowed such feature, but was mostly due to the extreme difficulty of describing the effects of actions under that premise. The state of a string was represented as a *list of crossings* (representing the various sections in which the string crossed a hole) and each loop in the string was identified by a pair of absolute positions in that list. This representation was compact, but actions on the strings could cause radical changes in the lists and their associated loops that, additionally, required renumbering. This meant a lack of elaboration tolerance since, very frequently an action affecting a local part of the puzzle might causes a substantial change in the representation of other unaffected parts of the puzzle. As a result, when a string crossed its own loops, there simply was no obvious way to describe the effects of an action in terms of formal changes in the list and the loops. The observed results on the list just showed an (apparently) unpredictable behaviour.

In the present paper, we overcome this difficulty and extend our framework to formalise the effect of actions that may deal with strings crossing their own loops. To this aim, we resort to our logical representation of string states introduced in (Cabalar and Santos, 2006) where the string is decomposed into segments treated as independent individuals that can be created and destroyed, forming or removing loops as a consequence. An important advantage of this representation is that segments that are not affected by the application of an action can be kept unchanged by application of the *inertia default* rule. To cope with the non-monotonicity of inertia, the original approach in (Cabalar and Santos, 2006) resorted to *First-Order Equilibrium Logic* (Pearce and Valverde, 2004) a complete logical characterisation of *Answer Set Programming* (ASP) (Gelfond and Lifschitz, 1988; Marek and Truszczyński, 1999; Niemelä, 1999), while the representation of time was based on *Situation Calculus* (McCarthy and Hayes, 1969). Our current formalisation also relies on ASP, but uses instead the linear-temporal extension of equilibrium logic introduced in (Aguado et al., 2013) that gave birth to the *Temporal ASP* framework (Cabalar et al., 2018) and its associated solver `telingo` (Cabalar et al., 2019). The advantage of this choice is that we can also test the effects of our segment-based formalisation in the computer, something that was only done for the list-based representation until now.

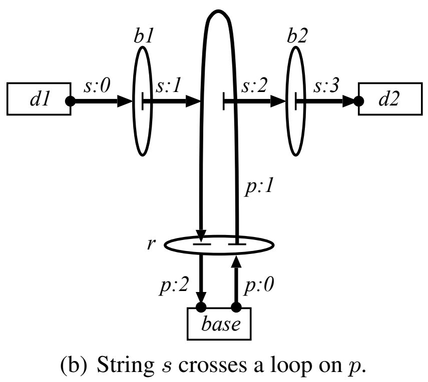
The main contribution of the paper is the formal characterisation of the effects of picking and pulling string segments through holes, regardless of whether the associated loops were created or destroyed on the same string or on different ones. This solution is general enough to be applied to various situations involving actions over string segments and holes. The consideration of actions related to winding (and unwinding) knots and the deployment of these ideas in real application domains is one of our major future work goals. This includes tasks such as autonomous needle steering (aiming at autonomous or semi-autonomous surgery) or the manipulation of (and reasoning about) real-world flexible objects by a collaborative robot in an industrial setting. However, applications require first a solid formalisation, and for that purpose, studying puzzles that contain enough complexity (as proposed by John McCarthy in the beginnings of KR (Morgenstern and McIlraith, 2011)) helps us to concentrate on the relevant aspects of the problem at hand. As a proof of concept, and for a better understanding of the technical material introduced in this work, we provide a formalisation of the solution for the so-called *African Ring* puzzle (Figure 3(a)) that involves loop manipulation on a unique string. To the best of our knowledge this is the first formal account of this domain.

2 Background

This paper uses some of the definitions introduced in our previous work (Cabalar and Santos, 2011; Santos and Cabalar, 2016; Cabalar and Santos, 2016): a hole H can be abstracted as a delimited surface with two faces, arbitrarily named H^+ and H^-, whereas a string S can be thought as an arbitrarily long linear structure with two tips, the negative S^- or *start*, and the positive S^+ or *end*. For any hole face F, we write $opp(F)$ to stand for its opposite face, that is, $opp(H^+) = H^-$ and $opp(H^-) = H^+$. In a given state, the string may be crossing a sequence of holes, from the start to the end of the string, that can be collected in a list, $chain(S)$. Each hole crossing in this list is represented by the exit hole face of the crossing. Although the state of the strings can be fully captured by these $chain$-lists, it will be usually convenient to depict their configuration using simplified diagrams. As an illustration, Figure 1(a) shows the initial state of the *Fisherman's Folly* puzzle (see (Cabalar and Santos, 2011)), whose goal is to remove a ring r from an entanglement of objects. Each holed object is represented as an ellipse whose "visible" face is the positive one. The puzzle has four holed objects, $b1, b2, ph$ and r. Strings are fragmented into their *segments* (portions delimited by crossings and tips) which, in their turn, are represented as thick arrows. The puzzle has two strings, s and p, the former consists of four segments, from $s:0$ to $s:3$, and the latter has only two, $p:0$ and $p:1$. Finally, rectangles represent regular, non-holed objects (in this case, $base, d1$ and $d2$) usually linked to some string tip. The puzzle states are represented by the chains $chain(s) = [b1^+, ph^+, b2^+]$ and $chain(p) = [r^+]$ together with the links among objects. In their initial for-



(a) String s crosses a holed object ph.



(b) String s crosses a loop on p.

Figure 1: Variants of the Fisherman's Folly puzzle.

malization, Cabalar and Santos (2011) exclusively studied the effects of action $pass(X, F)$, that is, passing an object X (linked to some string tip) through a given hole towards some of its faces F. An important simplification was that passing an object also pulled any linked string without representing intermediate states, something that prevented the formation of string loops. Under this assumption, Cabalar and Santos (2011) built a blind search Prolog planner that described the effects of $pass$ actions in terms of transformations on the $chain$-lists. Later on, Cabalar and Santos (2016) extended this setting to incorporate (a relevant type of) string loops. A *string loop* is defined as a subsequence of string segments started by a crossing towards a face F and ending by a crossing towards $opp(F)$. In other words, the loop starts crossing a hole H in some direction and ends crossing H back in the opposite direction.

Figure 1(b) shows a variant of the Fisherman's Folly puzzle where the holed object ph has been actually replaced by a string loop on p. In this variant, the chain for p becomes $chain(p) = [r^+, r^-]$ and the loop is formed by its second segment $p : 1$. The complexity introduced by string loops is that, as we can see in the figure, they also behave as holes. For instance, if we call the loop formed by $p : 1$ by the symbol l, then the string segment that lies in between $b1$ and $b2$ (in Figure 1(b)) can be represented[1] as $chain(s) = [b1^+, l^+, b2^+]$. As it can be imagined, string loops are difficult to formalise, since they can be dynamically created or removed, depending on the actions performed on the string. Moreover, as accounted in (Santos and Cabalar, 2016), they also form a hierarchy: a *subloop* may occur as a subsequence of a larger loop. When a subloop is destroyed, some of the interactions with other strings may affect its parent loop.

In (Cabalar and Santos, 2016), loops were named according to their relative position in the chain. For instance, in a chain of the form $chain(s) = [F_1, F_2, r^+, F_4, F_5, r^-, F_7]$, the loop formed on r was denoted as $l(s, [3, 6])$ meaning that we take the sublist of $chain(s)$ comprising from the third to the sixth element, that is $[r^+, F_4, F_5, r^-]$. The advantage of this notation is that each loop can be *uniquely identified* by exclusively inspecting the set of chain lists in a state. Using this representation, Cabalar and Santos (2016) developed an extension of the Prolog prototype that was capable of describing the effects of two basic kinds of action involving loops, *pick* and *pull* (we will explain and extend them later). As shown in (Santos and Cabalar, 2016), this sufficed to capture the sequence of transitions in the solution of the *Easy-does-it* puzzle (Figure 2), whose goal is also to extract a ring from an entanglement of strings and rigid objects. In spite of this success, this representation also has an important drawback. As the loop names are relative to the current set of chains, the description of the effects of a given action becomes extremely cumbersome, since not only can loops be destroyed or created but also, and most frequently, their names must be recomputed in terms of the whole set of resulting chains. This makes the Prolog code difficult to follow, compromising correctness. Moreover, the code was

[1]We use the right thumb rule to fix the positive side of a loop.

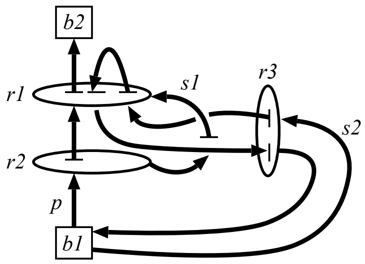


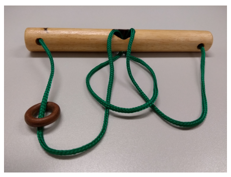
Figure 2: A state from the Easy-does-it puzzle.

built under the hard assumption that a string s could only cross loops on different strings, but *not loops formed by s itself.* This sufficed for solving the Easy-does-it puzzle, but became an assumption difficult to remove, since the changes in the loop numbering are always *global* with respect to the whole chain list, rather than *local*, affecting the part of the string altered by a given action.

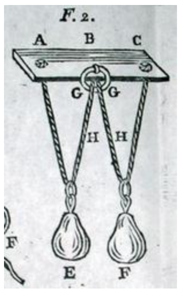
As we can see, the chain-based representation of loops suffered from an important lack of elaboration tolerance that prevented its application to scenarios where a single string may cross its own loops. One of such scenarios is the *African Ring* (or *African Ball*) puzzle, shown in Figure 3(a). This puzzle actually dates back to the 16th century, in its version known as the *Solomon's seal* in the literature (Figure 3(b)) (Pacioli, 2009; Rusca, 1743; dos Santos Hirth, 2015). The African Ring is composed of an entanglement of a single string fixed at the two ends of a rigid cylinder (the *post*) with a perforating hole in the middle, through which the string passes twice. The goal of this puzzle is to slide a ring (by means on non-destructive actions) that is initially hanging on one side of the string (see Figure 3(c)) to end up hanging instead on the other side of it, reaching a symmetric configuration (see Figure 3(d)). The main difficulty arises from the impossibility of passing the ring through the post hole. Equivalent configurations, like the Solomon's seal (see Figure 3(b)), may use a plate rather than a cylinder or include two hanging objects that are initially separated but must end up hanging together in the same side of the string.

3 Action Domains in Temporal ASP

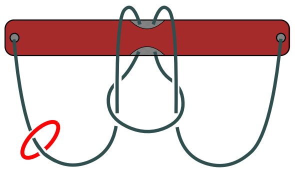
Our scenarios will be formally represented in a temporal extension of *Answer Set Programming* (ASP) (Gelfond and Lifschitz, 1988). In particular, we will use the input language of the temporal ASP tool `telingo` (Cabalar et al., 2019), a variant of the popular ASP solver `clingo` (Gebser et al., 2016) that incorporates operators and constructs for reasoning about finite traces in transition systems. Due to space limitations, we assume some familiarity with ASP and will just illustrate the temporal features of `telingo` through the examples. Our representation uses two main dynamic predicates: $h(F, V)$, meaning that fluent F holds value V; and $o(A)$ meaning that action A has occurred. We also use $c(F, V)$ to mean that fluent F was caused to get value V.



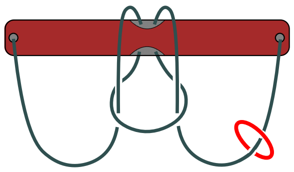
(a) African Ring



(b) Solomon's Seal.



(c) Initial state



(d) Goal state

Figure 3: African Ring Puzzle

```
1  #program dynamic.
2  c(gun,loaded) :-
3    o(load),'h(gun,unloaded).
4  c(turkey,dead) :-
5    o(shoot),'h(gun,loaded).
6  c(gun,unloaded):-
7    o(shoot),'h(gun,loaded).
8  h(F,V) :- c(F,V).
9  c(F) :- c(F,V).
10 h(F,V) :- 'h(F,V), not c(F).
11
12 #program initial.
13 h(gun,unloaded).
14 h(turkey,alive).
15
16 &tel{     &true ;> o(load)
17      ;> o(wait) ;> o(shoot)     }.
```

Listing 1: Yale Shooting Scenario in `telingo`.

Listing 1 shows a simple encoding of the well-known Yale Shooting scenario in `telingo`. Rules are organized in groups that follow the `#program` directive. For instance, lines 2-10 correspond to `dynamic` rules that affect all time points $t > 0$ in the temporal trace. Predicates preceded by `'` (read as "previous") refer to the situations at $t-1$ and all the rest to the situation at t. For instance, lines 2-3 assert that if we load the gun at t when it was unloaded at $t-1$; the gun is caused to be loaded at t. Similarly, lines 4-7 assert that shooting a gun that was loaded causes the turkey to be dead and the gun to be unloaded, respectively. Line 8 tells us that any caused value must also hold, whereas line 9 defines the abbreviation $c(F)$ to mean that fluent F has been caused for some value V. Line 10 is a compact and simple representation of the *inertia law*: if fluent F held value V before and we cannot prove it was caused (note the use of *default negation*) then it keeps the same value V. Lines 13-17 correspond to `initial` rules, that is, those that refer to the situation at $t = 0$. For instance, 13-14 assert that, initially, the turkey is alive and the gun is loaded. The construct `&tel` is used to introduce any temporal formula in the syntax of *Linear Temporal Logic* (Pnueli, 1977). For its use in this paper, it suffices to observe that `F ;> G` stands for the formula $F \wedge \bigcirc G$, that is, F holds now and G holds in the next state. Operator `;>` is right-associative: `F ;> G ;> H` stands for $F \wedge \bigcirc(G \wedge \bigcirc H)$. Thus, lines 16-17 assert that we load the gun at $t = 1$, then wait and then shoot. The `telingo` solver searches for answer sets using an iterative deepening strategy where the trace length (called *horizon*) is increased step by step until a solution is found. In our example, we obtain a unique temporal answer set where the gun is unloaded and the turkey is dead at $t = 3$ (among other facts).

4 A Segment-Based Encoding of String Loops

We proceed next to summarise the main aspects of our Temporal ASP formalisation for string theories. For space reasons, we will just show the main relevant parts of the `telingo` encoding: the whole package can be found in the `stringloops` public repository[2]. As said before, we use the African Ring puzzle (Figure 3(a)) as a running example. For simplicity, we consider the initial state in Figure 8 where the tips of the string cross the post hole rather than being linked to the post. In this way, we can ignore loops formed by linked objects (which is a simple elaboration) and concentrate on those formed by string passing holes. The puzzle solution is not affected, assuming that the string tips are not allowed to pass through the post hole. Our formalisation uses the same predicates $h(F,V)$, $c(F,V)$ and $o(A)$ seen in the previous section, where F is some fluent, V one of the values of F and A is some action. Each string segment is identified by a unique natural number. For instance, in Figure 8 we have segments 0 to 7. An integer fluent max will keep the maximum segment number used so far. A string consists of a sequence of segments and crossings that is captured by three main fluents, $start$, $next$ and $cross$. Given a string S and a segment X, $start(S)$ represents the starting segment of S, $next(X)$ represents the next segment of X and $cross(X)$ represents the outgoing hole face crossed by X. The value of $cross(X)$ is the outgoing hole face $f(H,D)$ of a crossing of the segment X towards the hole H, where D represents the face direction, positive $D = p$ or negative $D = n$. Lines 2-9 of Listing 2 show the set of facts corresponding to the initial state in Figure 8. It is worth to note that the string crosses twice one of its own loops, denoted as $l(4,4)$. This loop is formed because segment 4 comes from the previous crossing $cross(3)$ to $f(h,n)$ whereas $cross(4)$ goes toward the opposite hole face $f(h,p)$. In general, $l(X,Y)$ denotes a loop formed by two (possibly non-consecutive) crossings through the same hole but in opposite directions. These loops can be defined in terms of the fluents $next$ and $cross$ as shown in Listing 3. The predicate $connect(X,Y)$ is first defined as the transitive closure of $h(next(X),Y)$. Then, the two

[2]https://github.com/cabalar/stringloops.git

rules for the predicate $loop(X, Y)$ are self-evident: function `@opp(F)` just returns $opp(F)$. Using these rules on facts from Listing 2, we can actually derive five loops denoted as $l(4, 6), l(4, 4), l(3, 5), l(1, 6), l(1, 4)$. A loop can be included in a parent loop: for instance, loop $l(4, 4)$ is included in $l(4, 6)$ (which covers segments $4, 5$ and 6) and this, in its turn, is included in $l(1, 6)$ that, in this example, is a parent loop of all the rest.

```
 1 #program initial.
 2 h(start(s),0).
 3 h(next(0),1). h(cross(0),f(h,n)).
 4 h(next(1),2). h(cross(1),f(r,p)).
 5 h(next(2),3). h(cross(2),f(l(4,4),p)).
 6 h(next(3),4). h(cross(3),f(h,n)).
 7 h(next(4),5). h(cross(4),f(h,p)).
 8 h(next(5),6). h(cross(5),f(l(4,4),n)).
 9 h(next(6),7). h(cross(6),f(h,p)).
10
11 &tel{ &true
12   ;> o(slide(r,p))
13   ;> o(pick(4,f(h,p))) & o(embrace(5))
14   ;> o(pull(l(9,9))) ;> o(shrink)
15   ;> o(pull(l(4,4))) ;> o(shrink)
16   ;> o(slide(r,p))
17   ;> o(slide(r,p))
18   ;> o(pull(l(10,10))) ;> o(shrink)
19   ;> o(pull(l(12,12))) ;> o(shrink)
20   ;> o(pull(l(3,3))) ;> o(shrink)
21   ;> o(slide(r,p))
22 }.
```

Listing 2: Initial state and sequence of actions.

```
#program always.
connect(X,Y):-h(next(X),Y).
connect(X,Y):-h(next(X),Z),connect(Z,Y).

loop(X,Y) :- h(cross(W),F),h(next(W),X),
 h(cross(Y),F2),F2=@opp(F),connect(X,Y).
loop(X,X) :- h(cross(W),F),h(next(W),X),
 h(cross(X),F2), F2=@opp(F).
```

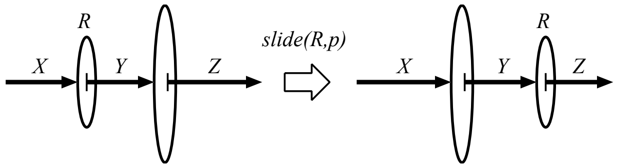
Listing 3: Definition of loops.

The African Ring can be solved performing the sequence of actions in lines 12-21 of Listing 2. As we explained before, in this work, we have focused on the simulation of these actions, since the precise formalisation of their effects is a challenging, still unsolved, problem. Obtaining this sequence of actions as a result of an efficient planning method is right now unfeasible if we directly use the current formalisation in `telingo`. However, an automated planning tool cannot be designed if we cannot even predict the result of a single transition step in a precise way. That was the situation before the current work: now a program can describe the result accurately. The domains considered in this work constitute a challenge for state-of-the-art planning systems (as pointed out in Santos and Cabalar (2013a)) since states must be represented by arbitrarily long lists or an arbitrarily growing set of segments. As a proof-of-concept, in (Cabalar and Santos, 2011), we proposed a planner for a simpler string-puzzle domain (without string loops) that solved the Fisherman's Folly puzzle. Efficient planning, however, constitutes a different research goal *per se*, where we plan to explore more efficient ASP encodings. This problem is left for future work.

Although in the `stringloops` repository we have encoded other kinds of actions (like passing string tips and linked objects through holes), we will focus here on the three basic actions used in the African Ring solution: $slide$, $pick$ and $pull$.

4.1 The Slide Action

The action $slide(R, D)$ is applicable on a ring R crossed by a unique string S and performs a movement towards the positive or the negative tip of the string, i.e., $D \in \{p, n\}$. Figure 4 shows a diagram and a listing: both correspond to sliding the ring R towards the positive tip of the string. We only



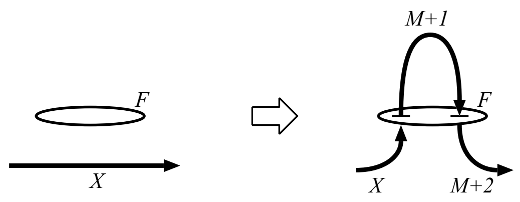
```
 1 #program dynamic.
 2 slided(D,X,f(R,E)) :-
 3   o(slide(R,D)), 'h(cross(X),f(R,E)).
 4 slided(D,X) :- slided(D,X,_).
 5
 6 c(cross(X),F) :- slided(p,X),
 7     'h(next(X),Y), 'h(cross(Y),F).
 8 c(cross(Y),F) :- slided(p,X,F),
 9     'h(next(X),Y).
10
11 upd_end(Y,X):-slided(p,X),'h(next(X),Y).
12
13 upd_start(Z,Y) :- slided(p,X),
14     'h(next(X),Y), 'h(next(Y),Z).
15
16 remove(X):- slided(p,X),
17     'h(next(X),Y), 'end(S,Y).
18 c(next(X))   :- remove(X).
19 c(cross(X)) :- remove(X).
```

Figure 4: Sliding R towards the end of the string.

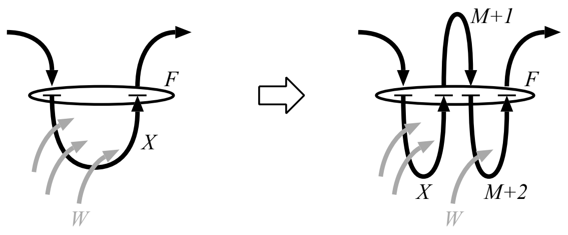
show a partial formalisation of this movement – the negative slide is similar. Lines 2-3 in the listing tells us that, when $slide(R, D)$ occurs, the predicate $slided(D, X, F)$ captures the slide direction D, the string segment X that was entering the ring and the outgoing ring face F. The predicate $slided(D, X)$ just collects the first two arguments. Note that, in the diagram, the sequence of segments formed by $X \rightarrow Y \rightarrow Z$ is maintained but their crossings are swapped. This last feature is captured by Lines 6-9. Lines 11-14 are used to rename those previous loops (referred in crossings) that may be affected by this change. For instance, any loop of the form $l(S, Y)$ should become now $l(S, X)$ since the old segments X and Y become now X altogether. Predicate $upd\_end(Y, X)$ is used to perform this change (we omit its encoding). Similarly, $upd\_start(Z, Y)$ renames all loops of the form $l(Z, E)$ into $l(Y, E)$. Finally, lines 16-19 cover the case in which Y is the last segment of the string. When this happens, we remove all facts for $next(X)$ and $cross(X)$ by releases them from inertia, so that Y will not be referred any more and X will become the last string segment. Figure 9(a), shows the result of applying $slide(r, p)$ on the African Ring's initial situation in Figure 8. The ring r (the small ellipse) is slided towards the end of the string and passes through loop $l(4, 4)$, which was the next crossing in that direction.

4.2 The Pick Action

The next action used in the African Ring puzzle is *picking* the string. The action $pick(X, F)$ passes some arbitrary point inside the segment X towards the hole face F creating a new loop. Figure 5(a) contains the corresponding diagram for this action. As we can see, picking creates two new segments, $M+1$ and $M+2$, where M was the previous maximum number used as segment identifier (as we saw, this number is captured by fluent max). The creation



(a) Segments and crossings created by $pick(X, F)$.



(b) Effects of $pick(X, F)$ and $embrace(W)$.

Figure 5: Effects of action $pick$.

of these segments and crossings follows a similar pattern as lines 6-9 of Figure 4, so we omit that part of the encoding. As happened with $slide$, some loop names may be affected by these changes. In particular, we include the rule

```
upd_end(X,M+2) :- o(pick(X,F)),
        'h(cross(X),F1), F!=F1, 'h(max,M).
```

that, when X was not crossing towards F, renames any loop $l(A, X)$ to $l(A, M+2)$, since the previous segment X has been decomposed now into the sequence $X \rightarrow M+1 \rightarrow M+2$. The most complicated situation we may face when picking is when X was part of a loop $l(A, B)$ and the loop end was crossing towards F. When this happens, three new loops are created: $l(A, X)$, $l(M+1, M+1)$ and $l(M+2, B)$ (if $B \neq X$) or $l(M+2, M+2)$ if $B = X$. For instance, Figure 5(b) shows a simple case where X is part of the loop $l(X, X)$ that is already crossing towards F. In this case, the three resulting loops are (the new) $l(X, X)$, $l(M+1, M+1)$ and $l(M+2, M+2)$. The complication here comes when we have one or more segments crossing that loop: it is unclear which of the new loops, the first one or the last one, will "inherit" those crossings. Figure 5(b) shows one of the three crossing segments called W that has been assigned to the loop $l(M+2, M+2)$ while the other two remain in $l(X, X)$. To remove this uncertainty, we combine action $pick(X, F)$ with a second action $embrace(W)$ that may be applied on several segments and points out which of them will be assigned to the last loop. Listing 4 shows the effects of picking when X is inside some loop crossed by other segments.

```
pickcross(W,A,B,D) :- o(pick(X,F)),
    'inloop(X,A,B),'h(cross(B),F),
    'h(cross(W),f(l(A,B),D)).

% is assigned to l(A,X), if not embraced
c(cross(W),f(l(A,X),D)) :- o(pick(X,_)),
  pickcross(W,A,_,D), not o(embrace(W)).

% is assigned to l(M+2,...) if embraced
c(cross(W),f(l(M+2,B),D)) :-
  o(pick(X,_)),
  pickcross(W,A,B,D), B!=X,
  o(embrace(W)), 'h(max,M).
c(cross(W),f(l(M+2,M+2),D)) :-
  o(pick(X,_)),
  pickcross(W,A,X,D),
  o(embrace(W)), 'h(max,M).
```

Listing 4: Effects of $pick(X, F)$, if X is in a crossed loop.

Finally, the diagram in Figure 9(b) is the result of performing, on the state shown in Figure 9(a), the actions $pick(4, f(h, p))$ and $embrace(5)$ simultaneously. Note that we pick segment $X = 4$ that was also a loop crossed by segments 2 and 5.

4.3 The Pull Action

The third action we use in the solution of the African Ring is pulling a loop back through its originator hole. The action $pull(L)$ is exclusively applied on some existing loop L. Pulling can be seen as the opposite action of picking: for instance, if we execute $pull(l(M+1, M+1))$ in the state of the diagram on the right in Figure 5(a) we obtain the state in the left diagram, where the loop has been destroyed (note that picking always creates a new loop). The effects of this action constitute the most elaborated part of the present formalisation. To avoid introducing errors or missing possible cases, we have separated each pull action into two transitions: in a first step, the execution of $pull(L)$ creates new elements as required, but may also possibly remove existing crossings, leaving their associated segments untouched. As a result, we may have sequences of segments only linked by the fluent $next$ but without actually passing through any crossing using fluent $cross$. In a second, consecutive step, we perform what we have called a $shrink$ operation. This fictitious action exclusively takes care of each sequence of segments without crossings, and joins them together in a unique segment.

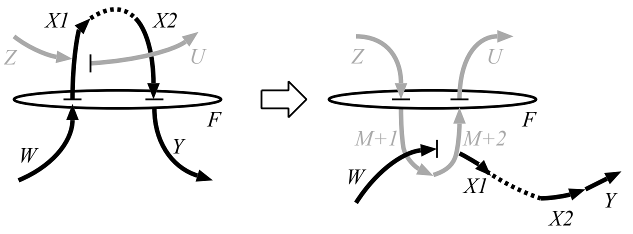
A first almost trivial case of $pull(L)$ is when L is not crossed at all. When this happens, we just remove the two crossings that originated the loop, by releasing them from inertia through predicate $c(\cdot)$. As we mentioned before, the action $shrink$ will then join the resulting consecutive segments. To illustrate this behaviour, take the application of $pull(l(3,3)); shrink$ on Figure 10(c) to produce Figure 10(d). The action $pull(l(3,3))$ just removes the two crossings through hole h leaving the sequence of segments $16 \to 3 \to 14$ without any intermediate crossing. After that, $shrink$ collapses this sequence into the unique segment 16 and updates the whole state accordingly.

```
crossedby(Z,D) :-
   'h(cross(Z),f(L,D)), o(pull(L)).
crossed :- crossedby(_,_).
c(cross(W)) :- not crossed,
   o(pull(l(X1,_))), 'h(next(W),X1).
c(cross(X2)) :- not crossed,
   o(pull(l(_,X2))).
```

Listing 5: Pulling an uncrossed loop.

When the pulled loop is crossed by other segments, the situation is much more complicated. The problem in this case is that these crossing segments will be indirectly picked through the hole originating the loop. For simplicity, we have only formalised $pull(L)$ when L is crossed by a unique segment. This simplified action suffices to formalise the most general case, since we can decompose any pull by a sequence of several picking and embracing actions. For instance, in Figure 9(a), we would normally pull the whole loop $l(4,4)$ towards the positive (upwards) face of h. However, this loop is crossed by two segments 2 and 5 and, thus, we decompose this movement into three actions: we first pick segment 4 towards $f(h,p)$ embracing 5; then we pull the right loop $l(9,9)$; and finally, we pull the loop the left loop $l(4,4)$. In this way, we can see Figure 9(d) as the overall result of pulling $l(4,4)$ on Figure 9(a). The direct effects of pulling a crossed loop are described in Figure 6 that contains both the rules and a diagram describing their meaning. In the diagram, the picked string is coloured in grey. Lines 2-3 create the new loop in the picked string formed by two new segments, $M+1$ and $M+2$. We use of two segments rather than a single one because one of the possible effects is that the picked string inherits a crossing through a parent loop, as we will see later on. The rest of the rules capture the transformations shown in the diagram. Segments $X2$ and Y will eventually collapse when we execute action $shrink$.

Finally, the action $pull(L)$ may produce effects on the other loops that are related to L. These effects are partially shown in Listing 6. One of those situations is when L has a parent loop, as described in lines 2-7 and depicted in detail in Figure 7. The most relevant effect in this case is that



```
1  c(max,M+2) :- crossed, 'h(max,M).
2  c(next(M+1),M+2) :- crossed, 'h(max,M).
3
4  % Pulled string
5  c(cross(W), f(l(M+1,M+2),D)) :-
6    o(pull(l(X1,_))), 'h(next(W),X1),
7    crossedby(Z,D), 'h(max,M).
8  c(cross(X2)) :- o(pull(l(_,X2))),
       crossed.
9
10 % Picked string
11 c(next(Z),M+1):-
12   crossedby(Z,_),'h(max,M).
13 c(cross(Z),F) :- crossedby(Z,_),
14   o(pull(l(_,X2))),
15   'h(cross(X2),F), 'h(max,M).
16 c(next(M+2),U) :- crossedby(Z,_),
17   'h(next(Z),U), 'h(max,M).
18 c(cross(M+2),F1) :- crossed,
19   o(pull(l(_,X2))),
20   'h(cross(X2),F),F1=@opp(F),'h(max,M).
```

Figure 6: Direct effects of pulling a crossed loop.

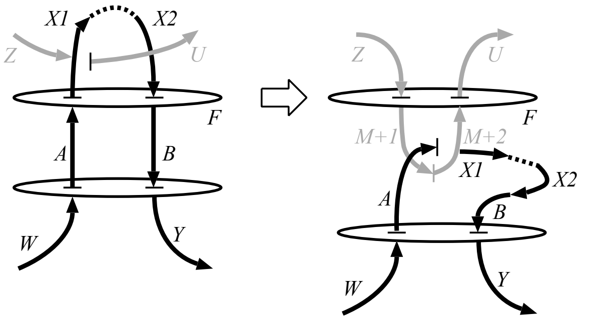


Figure 7: Pulling $l(X1,X2)$ with parent loop $l(A,B)$.

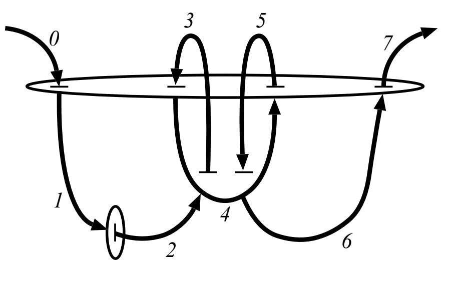


Figure 8: Initial state of the African Ring.

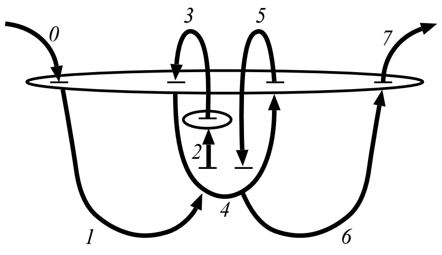
the picked string *inherits* a crossing through the parent loop $l(A, B)$. A second situation that affects other loops is when L is pulled in a situation where there is an adjacent loop on its left or on its right. In the listing, we only show the effects on a left adjacent loop – right adjacent loops are treated symmetrically. To understand the situation in a simple way, imagine that we follow backwards segment W in the diagram in Figure 6 and we find out that the string actually comes from the same hole, forming a loop $l(A, W)$ adjacent to $l(X1, X2)$ on its left. If $l(A, B)$ has no parent loops, and some segment was crossing $l(A, W)$, this crossing will disappear, since the loop is destroyed: this is captured by rule in lines 10-13. On the other hand, if $l(A, B)$ was inside some parent loop P, then any crossing through $l(A, W)$ must be replaced by the parent loop P in the whole state. This is done in lines 16-21 that use the auxiliary predicate upd_loop. To conclude this section, we include in Figures 9 and 10 the complete sequence of states generated by the execution in Listing 2, starting from the initial state displayed in Figure 8. As commented before, we assume that each $pull$ action is always followed by a $shrink$ action to remove possible sequences of segments without intermediate crossings.

```
 1 % Parent loop
 2 c(cross(M+1),f(l(A,B),D)) :-
 3     o(pull(l(X1,X2))), crossed,
 4     'parentloop(l(X1,X2),l(A,B)),
 5     'h(cross(X2),f(H,_)),
 6     'h(cross(B),f(H2,_)), H!=H2,
 7     crossedby(_,D), 'h(max,M).
 8
 9 % l(A,W) left adjacent loop, no parent
10 c(cross(Z)) :- o(pull(l(X1,_))),
11     'h(next(W),X1), 'leafloop(l(A,W)),
12     not 'hasparent(l(A,W)),
13     'h(cross(Z),f(l(A,W),_)).
14
15 % Left adjacent loop with parent P
16 upd_loop(l(A,W),P) :- o(pull(l(X1,_))),
17     'h(next(W),X1), 'leafloop(l(A,W)),
18     'parentloop(l(A,W),P).
19
20 c(cross(W),f(L2,D)) :- upd_loop(L1,L2),
21     'h(cross(W),f(L1,D)).
```

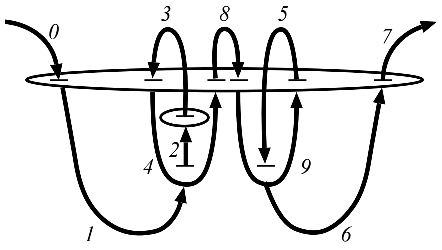
Listing 6: Effects of $pull(L)$ on other loops.

5 Related Work

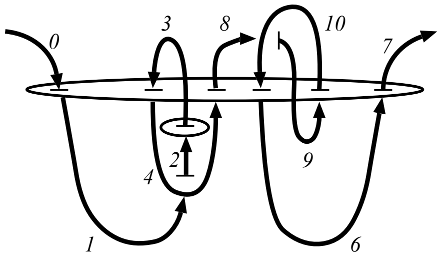
A recent up-to-date interdisciplinary survey of the investigation of knots from various disciplines is presented in (Santos, Cabalar, and Casati, 2019). Within this survey, only a few references can be considered as related to the work described in this paper. Strings and pins are used in (Freksa et al., 2018) as the medium for problem solving extending the range of possible solutions that can be obtained from using other tools, such as compass and straightedge. Three examples of spatial problem solving are given by Freksa et al. (2018): the construction of an ellipse, the solution for the shortest path problem and the angle trisection problem. The



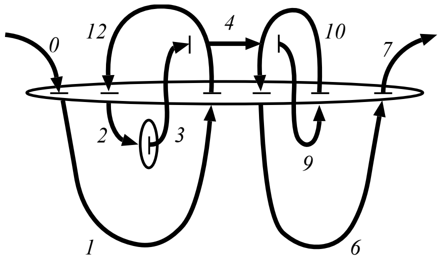
(a) After $slide(r, p)$.



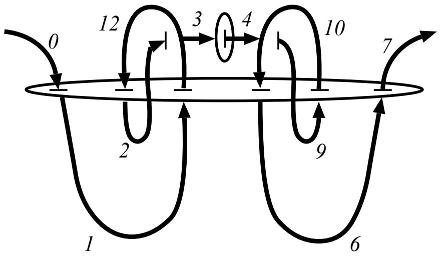
(b) After $pick(4, f(h, p))$, $embrace(5)$.



(c) After $pull(l(9, 9))$; $shrink$.

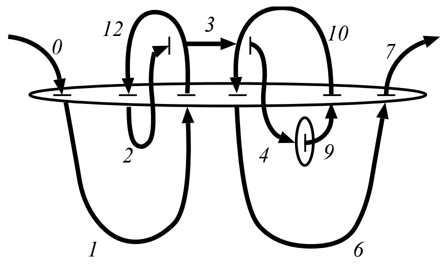


(d) After $pull(l(4, 4))$; $shrink$.

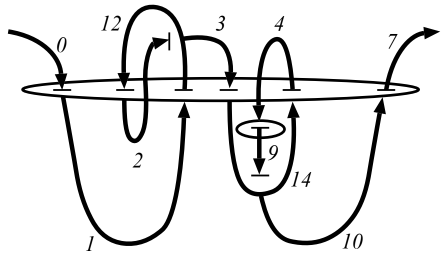


(e) After $slide(r, p)$.

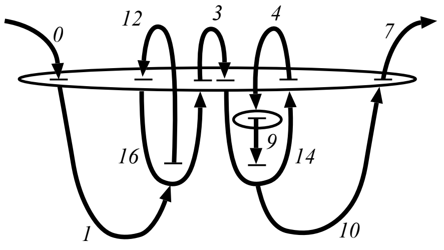
Figure 9: Formal solution for the African Ring puzzle (I).



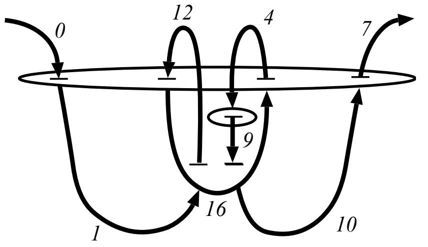
(a) After $slide(r, p)$.



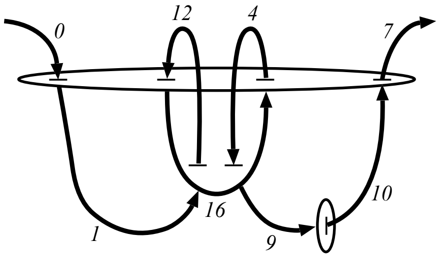
(b) After $pull(l(10, 10)); shrink$.



(c) After $pull(l(12, 12)); shrink$.



(d) After $pull(l(3, 3)); shrink$.



(e) After $slide(r, p)$.

Figure 10: Formal solution for the African Ring puzzle (II).

fundamental idea in that work was that spatial problem solving depends on the medium where the inferences are conducted, whereas, in the present paper we aim to abstract the medium by means of a formal representation. Closer to the kind of problems reported in the present work is the knot equivalence problem cited in (Turing, 1954), where states of spatial puzzles are represented as strings of symbols, and actions are substitutions of subsets of each of these symbols with other strings. Nevertheless, if the problem has a solution, there should be an algorithm capable of finding it. To the best of our knowledge, within the current automated reasoning literature, only our previous work has tackled this problem (Cabalar and Santos, 2006; Santos and Cabalar, 2008; Cabalar and Santos, 2011, 2016; Santos and Cabalar, 2016).

6 Conclusions

We have provided a formalisation of scenarios involving holed objects and strings, including string loops treated as holes that use the string segments as basic domain entities. A state is represented in terms of a pair of fluents that describe, for each segment, the next segment in the string and the next crossed hole, respectively. Using that representation, we have provided the axiomatisation of three basic actions, $slide$, $pick$ and $pull$, that are general enough to cover scenarios in which a string may cross its own loops, something not possible in the previous related formalisations. The axiomatisation has been encoded in the Temporal Answer Set Programming (ASP) tool `telingo` which has been used to describe the first formalisation of the solution to the African Ring problem, to the best of our knowledge.

In principle, the same temporal ASP encoding can also be used for planning, and not only for simulation of the effects. To do so, we must also include additional constraints describing allowed movements: for instance, typically, string tips may be linked to large objects that cannot pass through some holes. In the case of the African Ring puzzle, the ring itself cannot pass through any other holed object in the domain. However, our initial experiments for solving string planning problems with Temporal ASP have only succeeded on extremely small scenarios. Future work will explore a more efficient encoding for planning and the extension to more general or combined actions.

In a broader sense, research on the several aspects of the manipulation of (and reasoning about) flexible objects such as strings has a cross disciplinary interest. Beyond the far reaching set of applications of topological knot theory, the use of strings and ropes were one of the earliest kinds of tools used by the primitive hunters and gatherers (Santos, Cabalar, and Casati, 2019). We claim that reasoning about knots is a distinctive aspect of intelligence where cognition, knowledge representation and reasoning play a crucial role that cannot be replaced by pattern matching or statistical prediction algorithms. The actual development of robust knowledge representation and efficient problem solving strategies for this domain is still a largely unexplored area that deserves future investigations.

Acknowledgements

We wish to thank the anonymous reviewers for their helpful comments and suggestions. This work has been partially supported by MINECO (grant TIN2017-84453-P) and Xunta de Galicia (grant GPC ED431B 2019/03), Spain.

References

Aguado, F.; Cabalar, P.; Diéguez, M.; Pérez, G.; and Vidal, C. 2013. Temporal equilibrium logic: a survey. *Journal of Applied Non-Classical Logics* 23(1-2):2–24.

Cabalar, P., and Santos, P. 2006. Strings and holes: an exercise on spatial reasoning. In *Proc. of IBERAMIA*, volume 4140 of *LNAI*. Springer. 419–429.

Cabalar, P., and Santos, P. E. 2011. Formalising the Fisherman's Folly puzzle. *Artificial Intelligence* 175(1):346–377.

Cabalar, P., and Santos, P. E. 2016. A qualitative spatial representation of string loops as holes. *Artificial Intelligence* 238:1 – 10.

Cabalar, P.; Kaminski, R.; Schaub, T.; and Schuhmann, A. 2018. Temporal answer set programming on finite traces. *Theory and Practice of Logic Programming* 18(3-4):406–420.

Cabalar, P.; Kaminski, R.; Morkisch, P.; and Schaub, T. 2019. telingo = ASP + time. In Balduccini, M.; Lierler, Y.; and Woltran, S., eds., *Logic Programming and Nonmonotonic Reasoning - 15th International Conference, LPNMR 2019, Philadelphia, PA, USA, June 3-7, 2019, Proceedings*, volume 11481 of *Lecture Notes in Computer Science*, 256–269. Springer.

dos Santos Hirth, T. W. N. 2015. Luca pacioli and his 1500 book de viribus quantitatis. Master's thesis, Universidade de Lisboa.

Freksa, C.; Barkowsky, T.; Dylla, F.; Falomir, Z.; Olteteanu, A.-M.; and van de Ven, J. 2018. Spatial problem solving and cognition. In Zacks J, T. H., ed., *Representations in Mind and World*. New York: Routledge. 156–183.

Gebser, M.; Kaminski, R.; Kaufmann, B.; Ostrowski, M.; Schaub, T.; and Wanko, P. 2016. Theory solving made easy with clingo 5. In Carro, M.; King, A.; Saeedloei, N.; and Vos, M. D., eds., *Technical Communications of the 32nd International Conference on Logic Programming, ICLP 2016 TCs, October 16-21, 2016, New York City, USA*, volume 52 of *OASICS*, 2:1–2:15. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

Gelfond, M., and Lifschitz, V. 1988. The stable models semantics for logic programming. In *Proc. of the 5th Intl. Conf. on Logic Programming*, 1070–1080.

Kauffman, L. H. 2005. The mathematics and physics of knots. *Reports on Progress in Physics* 68(12):2829.

Marek, V., and Truszczyński, M. 1999. *Stable models and an alternative logic programming paradigm*. Springer-Verlag. 169–181.

McCarthy, J., and Hayes, P. J. 1969. Some philosophical problems from the standpoint of artificial intelligence. In *Machine Intelligence*, 463–502. Edinburgh University Press.

McCarthy, J. 1959. Programs with common sense. In *Proc. of the Teddington Conference on Mechanization of Thought Processes*, 75–91.

Menasco, William W.; Thistlethwaite, M., ed. 2005. *Handbook of Knot Theory*. Elsevier.

Morgenstern, L., and McIlraith, S. A. 2011. John mccarthy's legacy. *Artificial Intelligence* 175(1):1 – 24. John McCarthy's Legacy.

Niemelä, I. 1999. Logic programs with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence* 25:241–273.

Pacioli, L. 2009. *De viribus quantitatis*. Aboca.

Pearce, D., and Valverde, A. 2004. Towards a first order equilibrium logic for nonmonotonic reasoning. In Alferes, J. J., and Leite, J. A., eds., *Logics in Artificial Intelligence, 9th European Conference, JELIA 2004, Lisbon, Portugal, September 27-30, 2004, Proceedings*, volume 3229 of *Lecture Notes in Computer Science*, 147–160. Springer.

Pnueli, A. 1977. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science*, 46–57. IEEE Computer Society Press.

Reidemeister, K. 1983. *Knot Theory*. BCS Associates.

Rusca, P. 1743. *Il maestro de' giuochi piacevoli per uso delle civili conversazioni ornato con figure in Rame, con alcuni quesiti aritmetici, ed una regola facile per descrivere orologgj a sole orizontali*.

Sanchez, J.; Corrales, J.-A.; Bouzgarrou, B.-C.; and Mezouar, Y. 2018. Robotic manipulation and sensing of deformable objects in domestic and industrial applications: a survey. *The International Journal of Robotics Research* 37(7):688–716.

Santos, P. E., and Cabalar, P. 2008. The space within Fisherman's Folly: Playing with a puzzle in mereotopology. *Spatial Cognition & Computation* 8(1-2):47–64.

Santos, P., and Cabalar, P. 2013a. An investigation of actions, change and space. In *ICAPS 2013 - Proceedings of the 23rd International Conference on Automated Planning and Scheduling*, ICAPS 2013 - Proceedings of the 23rd International Conference on Automated Planning and Scheduling, 484–485. null ; Conference date: 10-06-2013 Through 14-06-2013.

Santos, P. E., and Cabalar, P. 2013b. An investigation of actions, change, space within a hole-loop dichotomy. In *Proc. of the 11th Intl. Symp. on Logical Formalizations of Commonsense Reasoning (Commonsense'13)*.

Santos, P. E., and Cabalar, P. 2016. Framing holes within a loop hierarchy. *Spatial Cognition & Computation* 16(1):54–95.

Santos, P. E.; Cabalar, P.; and Casati, R. 2019. The knowledge of knots: an interdisciplinary literature review. *Spatial Cognition & Computation* 19(4):334–358.

Turing, A. M. 1954. Solvable and unsolvable problems. *Science News* 31:7–23.