# Augmenting Stream Constraint Programming with Eventuality Conditions

Jasper C.H. Lee[1], Jimmy H.M. Lee[2], and Allen Z. Zhong[2]

[1] Department of Computer Science
Brown University, Providence, RI 02912, USA
`jasperchlee@brown.edu`

[2] Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
`{jlee,azhong}@cuhk.edu.hk`

**Abstract.** Stream constraint programming is a recent addition to the family of constraint programming frameworks, where variable domains are sets of infinite streams over finite alphabets. Previous works showed promising results for its applicability to real-world planning and control problems. In this paper, motivated by the modelling of planning applications, we improve the expressiveness of the framework by introducing 1) the "until" constraint, a new construct that is adapted from Linear Temporal Logic and 2) the `@` operator on streams, a syntactic sugar for which we provide a more efficient solving algorithm over simple desugaring. For both constructs, we propose corresponding novel solving algorithms and prove their correctness. We present competitive experimental results on the Missionaries and Cannibals logic puzzle and a standard path planning application on the grid, by comparing with Apt and Brand's method for verifying eventuality conditions using a CP approach.

## 1 Introduction

Stream constraint programming [11, 12] is a recent addition to the family of constraint programming frameworks. Instead of reasoning about finite strings [7], the domain of the constraint variables in a *Stream Constraint Satisfaction Problem* (St-CSP) consists of *infinite streams* over finite alphabets. A St-CSP solver computes not only one but *all* stream solutions to a given St-CSP, succinctly represented as a deterministic Büchi automaton. Because of the infinite stream domains, and the fact we can find all solutions, the framework is particularly suitable for modelling problems involving time series, for example in control and planning, using one variable for each stream as opposed to using one variable per stream per time point in traditional finite domain constraint programming [1]. Lallouet et al. [11] first demonstrated such capabilities by implementing the game controller of Digi Invaders[3], a popular game on vintage Casio calculator models,

[3] See `https://www.youtube.com/watch?v=1YafgAcmov4` for a video of the game as implemented by Casio.

using the St-CSP framework. Lee and Lee [12] further applied the framework to synthesise PID controllers for simple robotic systems[4].

In addition to using St-CSPs for control, Lee and Lee also proposed a framework for modelling planning problems as St-CSPs, adapting that of Ghallab et al. [6] for finite domain constraint programming. Even though the St-CSP framework can express the entirety of what finite domain CSPs could, there are still natural constraints on plans that we expect to be able to express but are unable to. For example, we cannot express the constraint that the generated plan must *eventually* satisfy a certain condition, without imposing a hard upper bound on the number of steps before the plan must satisfy the condition.

This paper focuses on enhancing the expressiveness of the St-CSP framework, using planning problems as a motivation. We introduce the "until" constraint (Section 3), adapted from Linear Temporal Logic (LTL) [14], which includes as a special case the "eventually" constraint. In addition, in the case where we do wish to concretely bound the number of steps before a condition is satisfied, we introduce the `@` operator (Section 4) to simplify the modelling from the approach of Lee and Lee. There are two advantages to using the new operator in constraints: 1) we can better leverage known structure to accelerate solving, and 2) the notation is significantly less cumbersome, as measured in the length of the constraint expressions. We give experimental evidence (Section 5) of the competitiveness of our new solving algorithms.

For space reasons, we give only proof sketches of some of the results. The full paper is available at `https://arxiv.org/abs/1806.04325`, which includes all proofs, constraint models of our experiments and also more detailed exposition.

## 2 Background

We review the basics of stream constraint programming.

**Existing Stream Expressions and Constraints.** A *stream* $a$ over a (finite) alphabet $\Sigma$ is a function $\mathbb{N}_0 \to \Sigma$. For example, the function $a(n) = n \bmod 2$ is a stream over *any* alphabet containing $\{0, 1\}$. The set of all streams with alphabet $\Sigma$ is denoted by $\Sigma^\omega$. The notation $a(i, \infty)$ is used for the stream suffix $a'$ where $a'(j) = a(j + i)$. For a language $L$, we similarly define $L(i, \infty) = \{a(i, \infty) \mid a \in L\}$. In this paper, we are only concerned with St-CSPs whose variables take alphabets that are integer intervals, i.e. $[m..n]^\omega$ for some $m \leq n \in \mathbb{Z}$. However, the framework generalises naturally to any other finite alphabets.

To specify expressions, there are primitives such as variable streams, which are the variables in the St-CSP, and constant streams. For example, the stream 2 denotes the stream $s$ where $s(i) = 2$ for all $i \geq 0$.

*Pointwise* operators, such as integer arithmetic operators {`+`, `-`, `*`, `/`, `%`}, combine two streams at each index using the corresponding operator. Integer arithmetic relational operators are {`lt`, `le`, `eq`, `ge`, `gt`, `ne`}. They compare two streams

[4] See `http://www.youtube.com/watch?v=dT56qAZt8hI` and `http://www.youtube.com/watch?v=5GvbG3pNOvY` for video demonstrations.

pointwisely and return a *pseudo-Boolean stream*, that is a stream in $[0..1]^\omega$. Pointwise Boolean operators {`and`, `or`} act on any two pseudo-Boolean streams $a$ and $b$. The final pointwise operator supported is `if-then-else`. Suppose $c$ is pseudo-Boolean, and $a, b$ are streams in general, then (`if` $c$ `then` $a$ `else` $b$)$(i)$ is $a(i)$ if $c(i) = 1$ and $b(i)$ otherwise. There are also three *temporal* operators, in the style of the Lucid programming language [16]: `first`, `next` and `fby`. Suppose $a$ and $b$ are streams. We have `first` $a$ being the constant stream of $a(0)$, and `next` $a$ being the "tail" of $a$, that is `next` $a = a(1, \infty)$. In addition, $a$ `fby` $b = c$ is the concatenation of the head of $a$ with $b$ ($a$ *followed by* $b$), that is $c(0) = a(0)$ and $c(i) = b(i-1)$ for $i \geq 1$. Note that stream expressions can involve stream variables. For example, (`first` $y$) `+` (`next` $x$) is an expression.

Given stream expressions, we can now use the following relations to express stream constraints. For integer arithmetic comparisons $R \in$ {`<`, `<=`, `==`, `>=`, `>`, `!=`}, the constraint $a\,R\,b$ is *satisfied* if and only if the arithmetic comparison $R$ is true at every point in the streams. Therefore, a constraint is *violated* if and only if there exists a time point at which the arithmetic comparison is false. For example, `next` $x$ `!=` $y$ `+` 1 is a constraint enforcing that the stream expression $y$ `+` 1 is not equal to the stream `next` $x$ at all time points. Similarly, we define the constraint $a$ `->` $b$ to hold if and only if for all $i \geq 0$, $a(i) \neq 0$ implies $b(i) \neq 0$. Here we use the C language convention for interpreting integers as Booleans.

Care should be taken to distinguish between constraints and relational expressions. Relational operators take two streams and output a pseudo-Boolean stream. Constraints, however, are relations on streams. Two simple examples illustrate the difference: $x$ `le` 4 is a pseudo-Boolean stream, whereas $x$ `<=` 4 is a constraint that enforces $x$ to be less than or equal to 4 at every time point.

**Stream Constraint Satisfaction Problems.**

**Definition 1.** *[11, 12] A* stream constraint satisfaction problem *(St-CSP) is a triple $P = (X, D, C)$, where $X$ is the set of variables and $D(x) = (\Sigma(x))^\omega$ is the* domain *of $x \in X$, the set of all streams with alphabet $\Sigma(x)$. A constraint $c \in C$ is defined on an ordered subset $Scope(c)$ of variables, and every constraint must be formed as specified previously (though it is the aim of this paper to extend the class of specifiable constraints).*

Fig. 1 gives an example St-CSP. An *assignment* $A : X \to \bigcup_{x \in X} D(x)$ is a function mapping a variable $x_i \in X$ to an element in its domain $D(x_i)$. A constraint $c$ is satisfied by an assignment $A$ if and only if it is satisfied by the streams $\{A(x)\}_{x \in Scope(c)}$, and a St-CSP $P$ is *satisfied* by $A$ if and only if all constraints $c \in C$ are satisfied by $A$. We call the assignment $A$ a *solution* of the St-CSP $P$. We denote the *solution set* of $P$, namely the set of all solutions $A$ to $P$, by $sol(P)$. The St-CSP $P$ is *satisfiable* if $sol(P)$ is non-empty, and *unsatisfiable* otherwise. We also say that two St-CSPs $P$ and $P'$ are *equivalent* (denoted $P \equiv P'$) when $sol(P) = sol(P')$.

Given a set of constraints $C$ and an integer $i$, the *shifted view* of $C$ is defined as $C(i, \infty) = \{c_k(i, \infty) \mid c_k \in C\}$ by interpreting constraints as languages. Similarly, given an St-CSP $P = (X, D, C)$ and a point $i$, the *shifted view* of $P$ is defined as $\hat{P}(i) = (X, D, C(i, \infty))$.

**Solving St-CSPs.** Lallouet et al. [11] showed that the solution set $sol(P)$ of a St-CSP $P$ is a *deterministic $\omega$-regular language*, accepted by some *deterministic Büchi automaton* $\mathcal{A}$, which is a deterministic finite automaton for languages of streams [3]. A stream $s$ is accepted by $\mathcal{A}$ if the execution of $\mathcal{A}$ on input $s$ visits accepting states of $\mathcal{A}$ infinitely many times. When given a St-CSP $P$, the goal of a St-CSP solver, then, is to produce a *deterministic Büchi automaton* $\mathcal{A}$, called a *solution automaton* of $P$, that accepts the language $sol(P)$. We note that the work of Golden and Pang [7] for finite string constraint reasoning also finds all solutions as a single regular expression.

A St-CSP can be solved by a two-step approach [11, 12]. First, a given St-CSP $P$ is *normalised* into some normal form $P'$ where auxiliary variables may be introduced, but $P'$ is equivalent to $P$ *modulo* the auxiliary variables. Afterwards, the *search tree* (as defined below) is explored and "morphed" into a deterministic Büchi automaton via a *dominance detection procedure*, which is then output as the solution automaton. In the rest of the paper, when we augment the language for specifying stream expressions and constraints, we shall also follow the above two-step approach to solve these new classes of St-CSPs. As such, we only have to (a) specify our new normal forms, (b) give a corresponding normalisation procedure, and (c) detail the new dominance detection procedures.

We now define the notion of search trees for St-CSPs, adapted from that for traditional finite-domain CSPs [4]. We also describe how they are explored and how dominance detection allows us to compute solution automata from search trees. A *search tree* for a St-CSP $P$ is a tree with potentially infinite height. Its nodes are St-CSPs with the root node being $P$ itself. The *level* of a node $N$ is defined as 0 for the root node and recursively for descendants. A child node $Q' = (X, D, C \cup \{c'\})$ at level $k + 1$ is constructed from a parent node $P' = (X, D, C)$ at level $k$ and an *instantaneous assignment* $\tau(x) \in \Sigma(x)$, where $\tau$ takes a stream variable $x$ and returns a value in $\Sigma(x)$. In other words, $\tau$ assigns a value to each variable at time point $k$. The constraint $c'$ specifies that for all $x \in X$, $x(k) = \tau(x)$ and for all $i \neq k$, $x(i)$ is unconstrained. We write $P' \xrightarrow{\tau} Q'$ for such a parent to child construction, and label the edge on the tree between the two nodes with $\tau$. During search in practice, we shall *not* consider every possible instantaneous assignment, but instead consider only the ones remaining after applying *prefix-k consistency* [11].

We can identify a search node $Q$ at level $k$ with the shifted view $\hat{Q}(k)$. Taking this view, if $\hat{P}(k) = (X, D, C)$ is the parent node of $\hat{Q}(k + 1)$, then $\hat{Q}(k + 1) = (X, D, C \cup \{c'\})(1) = (X, D, (C \cup \{c'\})(1, \infty))$ where $c'$ is the same constraint as defined above.

Recall that a constraint violation requires only a single time point at which the constraint is false. Therefore, we can generalise the definition of constraint violation such that a finite prefix of an assignment can violate a constraint. A sequence of instantaneous assignments from the root to a node is isomorphic to a finite prefix of an assignment, and so the definition again generalises. Suppose $F = (X, D, C)$ is a node at level $k$ such that $\{\tau_i\}_{i \in [1..k]}$ is the sequence of instantaneous assignments that constructs $F$ from the root node, i.e. $P \xrightarrow{\tau_1} \dots \xrightarrow{\tau_k} F$. We say node $F$ is a *failure* if and only if $\{\tau_i\}_{i \in [1..k]}$ violates a constraint $c \in C$.

Given a normalised St-CSP $P$, its search tree is then explored using depth first search. Backtracking happens when the current search node is a failure. A search node $M$ at level $k$ is said to *dominate* another search node $N$ at level $k'$, written $N \prec M$, if and only if their shifted views are equivalent ($\hat{M}(k) \equiv \hat{N}(k')$) and $M$ is visited before $N$ during the search [11, 12]. When the algorithm visits a search node $N$ that is dominated by a previously visited node $M$, the edge pointing to $N$ is redirected to $M$ instead. If the algorithm terminates, then the resulting (finite) structure is a deterministic Büchi automaton (subject to accepting states being specified). If dominance detection were perfect, then the search algorithm terminates, because every branch either ends in a failure or contains two nodes with the same shifted views [11]. The crucial missing detail from this high-level algorithm, then, is *exactly* how dominance is *detected* in practice. Search node dominance is an inherently semantic notion, implying that it is often inefficient to detect precisely. Thus, previous works identify efficient *syntactic approximations* to detecting dominance such that the overall search algorithm terminates [11, 12]. We shall also give a new dominance detection procedure in light of the new ways of forming stream expressions and constraints. As for specifying the set of accepting states, previous work take *all* states as accepting states, whereas we shall give a more nuanced criterion.

## 3 The "Until" Constraint

In this section, we introduce the "until" constraint to the St-CSP framework. Recall that all the stream constraints introduced in Section 2 are pointwise predicates. That is, the constraint is satisfied if its corresponding predicate holds for every single time point of its input streams. The "until" constraint, as we shall later see, is *not* a pointwise constraint.

Let us consider the following path planning problem on the standard $n \times n$ grid world domain [15, 8]. Between any two neighbouring vertices on the grid, there could be 0, 1 or 2 *directed* edges. We ask for all paths on the directed graph from a given start point that eventually visit a given end point.

Our method finds more than a shortest path. Modelling this problem as a St-CSP allows us to find a succinct description of *all* the paths, and moreover allows for additional side constraints. Well-studied side constraints in the literature include precedence constraints [10] and time window constraints [13].

We can formulate as a St-CSP the condition that the path starts at $(i_s, j_s)$, has to respect the graph, and furthermore in the St-CSP model check whether the goal of visiting the end point $(i_g, j_g)$ is attained. This St-CSP is shown in Fig. 1. We use variables $x, y$ to represent the $x$ and $y$ coordinates of the current position. In addition, a variable *goal* denotes if we have visited the end point. The second to last constraint is such that if *goal* is true in one time point, it stays true in the next one as well. The last constraint says that if the path has reached the end point, then it stays there indefinitely.

var $x, y$ with alphabet $[1..n]$
var *goal* with alphabet $[0..1]$

`first` $x$ `==` $i_s$
`first` $y$ `==` $j_s$

For each vertex $(i, j)$,
$(($`next` $x$ `eq` $i)$ `and` $($`next` $y$ `eq` $j))$ `->` $(($$x$ `eq` $i$ `and` $y$ `eq` $j)$ `or`
  $(x$ `eq` $i_1$ `and` $y$ `eq` $j_1)$ `or` $\ldots$ `or` $(x$ `eq` $i_{d_{(i,j)}}$ `and` $y$ `eq` $j_{d_{(i,j)}})$
where $(i_1, j_1), \ldots, (i_{d_{(i,j)}}, j_{d_{(i,j)}})$ have edges into $(i, j)$
and $d_{(i,j)}$ is the in-degree of $(i, j)$

*goal* `==` $(x$ `eq` $i_g$ `and` $y$ `eq` $j_g)$ `or` $(0$ `fby` *goal*$)$
*goal* `eq` $1$ `->` $((x$ `eq` `next` $x)$ `and` $(y$ `eq` `next` $y))$

Fig. 1: St-CSP Model for the Path Planning Problem

In this current model, we have not enforced that the goal is indeed *eventually* attained at some point. An undesirable solution to the St-CSP would be, for example, to stay in one location forever. However, variants of the "eventually" constraint is not expressible in the St-CSP framework prior to this work, since all constraints are inherently *pointwise.* Temporal operators are not expressive enough for our purpose, since these operators shift streams by a constant number of time points only. The "eventually" constraint, on the other hand, can be satisfied at an unbounded number of time points away into the future.

We thus introduce the "until" constraint, adapted from Linear Temporal Logic (LTL) [14] and essentially equivalent to "eventually" [5].

**Definition 2 (The "Until" Constraint).** *Given two streams $a, b$, the constraint $a$ `until` $b$ is satisfied if and only if there exists a time point $i \geq 0$ such that 1) for all $j < i$, $a(j) \neq 0$ and 2) $b(i) \neq 0$. We say that the constraint is* finally satisfied *at time point $i$ if $b(i) \neq 0$. Note that we are again adapting the C language convention for interpreting integers as Booleans.*

The "eventually" constraint is expressible in terms of the "until" constraint. Suppose we want to express the constraint that a predicate $G$ on stream elements eventually holds, for example if $G$ is "*goal* `eq` 1". Then, we can express the constraint as "1 `until` $G$", or in our particular example, "1 `until` (*goal* `eq` 1)". Conversely, "$a$ `until` $b$" is equivalent to "$c$ `==` $b$ `fby` (`next` $b$ `or` $c$); (`not` $c$) `->` ($a$ `ne` 0); `eventually` $b$;".

## 3.1 Normalising "Until" Constraints

In light of the "until" constraint, we give a new constraint normal form. A St-CSP is in *normal form* if it contains only constraints of the following forms:

- Primitive next constraints: $x_i$ `==` `next` $x_j$
- Primitive until constraints: $x_i$ `until` $x_j$
- Primitive pointwise constraints with no `next`, `fby` or `until` (but can contain `first` operators).

Any St-CSP can be transformed into this normal form by applying the rewriting system below. We adopt notations from programming language semantics theory [17], writing $c\,[\_]$ for *constraint contexts*, i.e. constraints with placeholders for syntactic substitution. For example, if $c\,[\_] = [\_ \texttt{ + 3 >= } 4]$, then $c\,[\texttt{first}\ \alpha] = [(\texttt{first}\ \alpha) \texttt{ + 3 >= } 4]$. We also write a constraint rewriting transition as $(C_0, C_1) \rightsquigarrow (C'_0, C'_1)$, where $C_0$, $C_1$, $C'_0$ and $C'_1$ are sets of constraints. $C_0$ is the set of constraints that potentially could be further normalised, and $C_1$ is the set that is already in normal form. Hence, the initial constraint pair for the St-CSP $(X, D, C)$ is $(C, \{\})$. Rules are applied *in arbitrary order* until none are applicable.

- $(C_0 \cup \{c\,[\texttt{next}\ \mathit{expr}]\}, C_1) \rightsquigarrow (C_0 \cup \{c\,[x_1], x_2 \texttt{ == } \mathit{expr}\}, C_1 \cup \{x_1 \texttt{ == next } x_2\})$, where $x_1$ and $x_2$ are fresh auxiliary stream variables.
- $(C_0 \cup \{c\,[\mathit{expr}_1\ \texttt{fby}\ \mathit{expr}_2]\}, C_1) \rightsquigarrow (C_0 \cup \{c\,[x_1], x_2 \texttt{ == } \mathit{expr}_1, x_3 \texttt{ == } \mathit{expr}_2\}, C_1 \cup \{\texttt{first}\ x_1 \texttt{ == first } x_2, x_3 \texttt{ == next } x_1\})$, where $x_1$, $x_2$ and $x_3$ are fresh auxiliary stream variables.
- $(C_0 \cup \{\mathit{expr}_1\ \texttt{until}\ \mathit{expr}_2\}, C_1) \rightsquigarrow (C_0 \cup \{x_1 \texttt{ == } \mathit{expr}_1, x_2 \texttt{ == } \mathit{expr}_2\}, C_1 \cup \{x_1\ \texttt{until}\ x_2\})$, where $x_1$ and $x_2$ are fresh auxiliary stream variables.

We can check easily the following properties of the new rewriting system.

**Proposition 1** *The new rewriting system always terminates, regardless of the order in which the rules are applied.*

**Proposition 2** *The rewriting system has the Church-Rosser property (up to auxiliary variable renaming).*

**Proposition 3** *The rewriting system is sound, in the sense that it preserves the projection of the solution set of the resulting St-CSP into the original variables.*

## 3.2 Search Algorithm and Dominance Detection

In the following, we assume that all given St-CSPs are in normal form.

Recalling the high-level solving algorithm in Section 2, we give in this section a concrete instantiation of the syntactic dominance detection procedure. Our syntactic procedure should possess two key properties. First, the procedure should be *sound*: if two search nodes are claimed to have equivalent shifted views by the procedure, then they do indeed have equivalent shifted views. Second, the approximation should be sufficiently close to the semantic notion, such that the overall search algorithm terminates and produces a *finite* structure. Otherwise, in the extreme scenario where the dominance detection procedure never reports any dominance, the search algorithm will simply search the entire (usually infinite) search tree, resulting in non-termination.

Our dominance detection procedure, as with previous works [11, 12], involves keeping track of a *syntactic* representation of the shifted view of each search node, and detects dominance by checking *syntactic equivalence* between the two representations. Hereafter, we refer to search nodes and their syntactic representations interchangeably for narratory simplicity. Each search node, then, is

**Algorithm 1** Dominance Detection with Until Constraints

```
1: function CONSTRUCT(Search Node $\hat{P}(k) = (C, h)$, Instantaneous Assignment $\tau$)
2:     Historic values $h' \leftarrow \emptyset$
3:     for all primitive next constraints $x_i$ == next $x_j$ do
4:         $h'(x_j) \leftarrow \tau(x_i)$
5:     Constraint Set $C' \leftarrow \emptyset$
6:     for all primitive until constraints $x_i$ until $x_j$ do
7:         if $\tau(x_j) = 0$ then
8:             $C' \leftarrow C' \cup \{x_i$ until $x_j\}$
9:     for all primitive pointwise, next constraints $c$ do
10:        Constraint $c' \leftarrow c$ evaluated with $\tau$
11:        if $c'$ is not a zeroth order tautology then
12:            $C' \leftarrow C' \cup \{c'\}$
13:    return $\hat{Q}(k+1) = (C', h')$
14: function AREEQUAL(Search Nodes $\hat{P}(k) = (C_P, h_P)$, $\hat{Q}(k') = (C_Q, h_Q)$)
15:    return $(C_P = C_Q) \wedge (h_P = h_Q)$
```

represented by two components: 1) a set $C$ of St-CSP constraints and 2) a table $h$, called *historic values*, storing for each variable $x_j$ in a primitive next constraint "$x_i$ `== next` $x_j$" the value assigned to $x_i$ *at the previous time point*. The historic values are used to enforce primitive next constraints. If a value $v$ is assigned to $x_i$ at the previous time point, then `first` $x_j$ `==` $v$ holds in the shifted view of the current search node. We thus store $v$ in the table entry for $x_j$.

Algorithm 1 gives pseudocode for two functions, CONSTRUCT and AREEQUAL, both adapted from the algorithm of Lee and Lee [12] with *minimal* changes (lines 6–8) to accommodate "until" constraints. The function CONSTRUCT takes a parent search node $\hat{P}(k)$ and an instantaneous assignment $\tau$, and outputs the corresponding child search node $\hat{Q}(k+1)$ (the new constraint set $C'$ and new historic values $h'$). The function AREEQUAL, on the input of two search nodes, just checks whether their components are syntactically equal.

We describe the function CONSTRUCT in more detail. The new set of historic values $h'$ is conceptually simple to compute. For each primitive next constraint "$x_i$ `== next` $x_j$", we store $h'(x_j) = \tau(x_i)$ where $\tau$ is the instantaneous assignment given for the construction of the child search node. The new constraint set $C'$ is computed from $C$ by processing each constraint individually: 1) For primitive next constraints, we keep them as is and put them into $C'$. 2) For primitive pointwise constraints, we follow Lee and Lee [12] in *evaluating* them using the instantaneous assignment $\tau$. That is, we substitute every variable stream $x$ appearing in an expression whose outermost operator is the `first` operator, using the value $\tau(x)$. This process produces expressions that consist entirely of constant streams, pointwise operators and `first` operators, and thus can be evaluated into a single constant stream. If, as a result, a primitive pointwise constraint becomes a numerical tautology (e.g. 1 `==` 1), we discard such a constraint. 3) For primitive until constraints "$x_i$ `until` $x_j$" (lines 6–8), we simply check whether

$\tau(x_j)$ is 1, namely if the constraint is satisfied by the instantaneous assignment $\tau$. If so, we discard the constraint; otherwise we keep it in $C'$.

When the search algorithm terminates, which provably happens as we shall state later, we have a finite automaton whose states have to be labelled as accepting or non-accepting. We choose the set of accepting states as those whose constraint set $C$ contains *no* primitive until constraints. As a special case, when the given St-CSP has no "until" constraints, then all the states are accepting.

We stress again that our algorithm requires minimal changes from previous work to support the use of "until" constraints in St-CSPs. The only changes we have are lines 6–8 for the treatment of primitive until constraints, as well as how we pick the set of accepting states.

We first show that the dominance detection procedure is sound. To do so, we show that from a parent search node $(C, h)$ and an instantaneous assignment $\tau$, Construct computes a child node $(C', h')$ representing the correct shifted view. Thus, if two search nodes are syntactically equivalent, the corresponding shifted views must also be equivalent.

**Theorem 1 (Soundness of dominance detection).** *Suppose the constraint set $C'$ of the shifted view of child node $\hat{Q}(k+1)$ is output by* Construct *from the constraint set $C$ of the parent node $\hat{P}(k)$ and the instantaneous assignment $\tau_k$. Then $sol(C' \cup \{c_2\}) = sol(\{c \cap \pi_{Scope(c)}(c_1) \mid c \in C\}(1,\infty))$ where $c_1$ is the constraint stating $x(0) = \tau_k(x)$ for all streams $x$, and $c_2$ is the constraint stating $x_j(0) = \tau_k(x_i)$ for all constraints $x_i$ `== next` $x_j$ in $C$ (and hence $C'$). Note that $c_2$ is enforced by the set of historic values $h'$ produced by* Construct.

*Proof.* (Sketch) The two solution sets share the same primitive pointwise constraints. Primitive next constraints in $C$ (and $C'$) are respected in both solution sets because of the constraint $c_2$. Primitive until constraints in $C$ are either preserved in $C'$ or removed by Construct depending on $\tau_k$. Hence the constraint sets are either both constrained by an until constraint or both are not.

Having analysed the dominance detection algorithm, we can leverage the results to prove termination and soundness of the overall search algorithm.

**Theorem 2 (Termination).** *Using this new dominance detection procedure, the search algorithm always terminates.*

*Proof.* (Sketch) Overall, the search algorithm can produce only finitely many syntactically distinct search nodes, and thus always terminates.

**Theorem 3 (Soundness and Completeness).** *The resulting solution automaton $\mathcal{A}$ accepts the same language $L(\mathcal{A})$ as the solution set $sol(P)$ of the input St-CSP $P$.*

*Proof.* (Sketch) $L(\mathcal{A}) \subseteq sol(P)$: The search algorithm ensures that primitive pointwise and next constraints are satisfied. Primitive until constraints are also satisfied by streams in the language by our choice of accepting states.
$sol(P) \subseteq L(\mathcal{A})$: Follows from Theorem 1 and induction on time points.

### 3.3 Automaton Pruning

As a post-processing step, we prune all states that cannot reach any accepting states via a flood-fill algorithm taking time linear in the size of the automaton (before pruning), which retains the accepted language by the following lemma.

**Lemma 1.** *Given a solution automaton $\mathcal{A}$, let $\mathcal{A}'$ be obtained from $\mathcal{A}$ by removing all states not reaching any accepting states. Then $L(\mathcal{A}) = L(\mathcal{A}')$.*

Furthermore, the pruning gives us the following guarantee about finite runs of the resulting automaton.

**Theorem 4.** *For any finite-length run of the generated and pruned solution automaton $\mathcal{A}$, corresponding to a finite string (stream prefix) $p$, there exists a solution stream $s \in L(\mathcal{A})$ such that $p$ is the prefix of $s$ of length $|p|$.*

*Proof.* (Sketch) Every finite run can be extended, inductively by the fact that each state can reach an accepting state.

Intuitively, the theorem says that, no matter how we run the automaton, we can always extend the (finite) stream prefix *generated so far* into an infinite-length solution stream. This therefore also guarantees that it is *sound* to generate solution streams by running the automaton.

We emphasise that this pruning is for soundness, not solving efficiency.

## 4 The `@` Operator

With the introduced "until" constraint along with a new solving algorithm, we can now model in St-CSPs conditions that need to be *eventually* satisfied. However, eventuality constraints might not be suitable for all application scenarios. It could be vital to be able to impose a strict upper bound on when a condition is satisfied, whilst with an eventuality constraint, the time at which a specified condition is satisfied could be arbitrarily far into the future.

Lee and Lee [12] propose using a constraint of the form "`first next` $\cdots$ `next` *goal* `==` 1" to model this bound, reflected by the number of `next` operators in the constraint as the time bound. There are, however, two disadvantages to this approach. First, such a constraint has its own structure that we could not exploit to improve solving if we were to simply use the above syntax and current solving algorithms. Second, the notation is cumbersome, with the length of the constraint scaling linearly with the upper bound we wish to impose. To remedy these two issues, we propose a new temporal operator "`@`" that acts as syntactic sugar, and further give another modification to the solving algorithm (more concretely, the dominance detection algorithm) to solve constraints involving the `@` operator efficiently. We note however that, since the `@` operator is simply a sugar, it does not enhance the expressiveness of the St-CSP framework.

**Definition 3 (The `@` operator).** *Given a stream $x$ (where $x$ is instantiated or is some expression even involving stream variables) and a* number $t \geq 1$, *the stream $x$*`@`*$t$ is defined as the constant stream $(x\texttt{@}t)(i) = x(t)$ for all $i \geq 0$. Equivalently, it is defined as* `first next` $\cdots$ `next` *$x$, where there are $t$ many* `next` *operators.*

We require that, for the purpose of this paper, the `@` operator to take only a concrete number, instead of a variable, for its second parameter $t$. Our solving algorithm relies crucially on this assumption.

### 4.1 Modified Constraint Normalisation

We first augment the constraint normal form to allow for primitive `@` constraints: $x_i$ `==` $x_j\texttt{@}t$, where $t \geq 1$.

Accordingly, we add the following rewriting rule to the constraint rewriting system presented in Section 3.

– $(C_0 \cup \{c\,[expr\texttt{@}t]\},\, C_1) \rightsquigarrow (C_0 \cup \{c\,[x_1],\, x_2 \texttt{ == } expr\},\, C_1 \cup \{x_1 \texttt{ == } x_2\texttt{@}t\})$, where $x_1$ and $x_2$ are fresh auxiliary stream variables.

This new rewriting system is also terminating, Church-Rosser and sound. The proofs are essentially identical to those in Section 3.

### 4.2 Changes to Dominance Detection

Having introduced the `@` operator, we adapt the function Construct by describing how primitive `@` constraints are modified when we construct a child search node from its parent. Given a primitive `@` constraint "$x_i$ `==` $x_j\texttt{@}t$" from a parent node, we consider two cases.

– If $t > 1$, then we include "$x_i$ `==` $x_j\texttt{@}(t-1)$" in the new constraint set.
– If $t = 1$, then we include "$x_i$ `==` `first` $x_j$" instead.

This modification is orthogonal to those for the "until" constraint. This new dominance detection procedure (namely Construct and AreEqual) is again sound, and induces a terminating, sound and complete overall search algorithm. The proofs are again essentially same as those in Section 3.

## 5 Experimental Results

We performed experiments in two settings to demonstrate the competitiveness of our approaches: 1) solving the Missionaries and Cannibals logic puzzle and 2) solving a standard path planning problem on grid instances. For each setting, we solve for plans that eventually attain the goal using the "until" constraint in the model, as well as for bounded-length plans using the `@` operator.

For the "until" experiments, we compare our approach to a standard CP approach proposed by Apt and Brand [1]. Their approach creates a *series* of finite domain CSPs, each corresponding to a finite horizon into the future, asking if the eventuality condition is satisfiable within the horizon. The time bound is incremented until the resulting CSP becomes satisfiable. (The idea was also used by van Beek and Chen [2], who credit Kautz and Selman [9].) As a result, if there is no upper bound a-priori on the minimum length of successful plans, this approach may not terminate. However, in the two settings we consider, such upper bounds do exist, and so we also experimented on using a CP solver to solve for a plan of exactly that length at the upper bound.

For the bounded-length plans scenario, we compare the use of the `@` operator to the use of the `first next` $\cdots$ `next` operator phrase, as well as to using a standard CP approach of solving the corresponding finite domain CSP.

All our experiments were run on an Intel Xeon CPU E5-2630 v2 (2.60GHz) machine with 256GB of RAM, with a timeout of 600 seconds. We used Gecode v6.0.0 as our finite domain CP solver. We also configured both the St-CSP solver and Gecode to *not* output the solutions to the file system, so as to minimise the impact of file I/O on time. The Gecode solver selects variables using the input order and according to the time point, which is the same as how the St-CSP solver label stream variables. Values are assigned the min value first. We tried fail-first for Gecode, but the results are less competitive.

## 5.1 Missionaries and Cannibals

In the Missionaries and Cannibals problem, there are $n$ missionaries and $n$ cannibals trying to cross a river from one bank to another, using a boat of capacity $b$ people. There are three constraints in this problem: 1) at any time, there could be at most $b$ people on the boat, 2) there must be at least one person on the boat on every trip and 3) for each bank, if there are any missionaries, then the cannibals cannot outnumber the missionaries; otherwise the missionaries will perish. The success condition is when everyone ends up on the other bank.

Table 1 shows the experimental results, when we solve using the St-CSP solver for *all* valid transportation plans that *eventually* attains the goal. Rows and columns in the table give different values of $n$ and $b$ respectively. Each entry in the table denotes the solving time in seconds for the test case. The results show that our solver is able to solve the problem for reasonably large instances without suffering from exponential increases in runtime.

We also performed experiments using the Apt and Brand framework [1] that uses traditional finite domain CP solvers. **Such CP approach timed out on all these instances.** On the other hand, for this particular problem there is, in fact, an upper bound on the number of steps of $n(b+1)$ if a feasible plan exists. We used a CP solver to solve for plans of such length, and because of the simple structure in the constraints, the solver was able to terminate under 15 seconds in all these instances, outperforming our approach.

The next set of experiments replaces the "until" constraint that *eventually* everyone is on the other bank with the condition that the goal must be satisfied at time $t$, which is a value we vary between test cases. Because the St-CSP model is modified, requiring different solving times, the range of parameters $(n, b)$ we experimented on is also different.

Table 1: Missionaries and Cannibals: "until"

| | $b = 4$ | $b = 5$ | $b = 6$ | $b = 7$ | $b = 8$ |
|---|---|---|---|---|---|
| $n = 40$ | 1.456 | 1.939 | 2.307 | 2.537 | 2.959 |
| $n = 60$ | 4.459 | 5.831 | 7.417 | 9.081 | 10.698 |
| $n = 80$ | 9.979 | 13.45 | 17.324 | 21.356 | 26.229 |
| $n = 100$ | 19.053 | 26.044 | 33.747 | 42.16 | 53.112 |
| $n = 120$ | 33.56 | 44.782 | 59.113 | 73.335 | 91.351 |
| $n = 140$ | 51.623 | 70.666 | 92.744 | 118.407 | 146.325 |
| $n = 160$ | 76.532 | 105.341 | 139.212 | 175.149 | 219.134 |
| $n = 180$ | 110.122 | 149.741 | 196.743 | 250.56 | 313.35 |
| $n = 200$ | 150.137 | 207.466 | 274.537 | 348.243 | 436.469 |
| $n = 220$ | 201.308 | 277.219 | 363.592 | 463.509 | – |
| $n = 240$ | 259.773 | 360.413 | 474.005 | – | – |

Table 2: Missionaries and Cannibals: Time bounded

(a) `@` vs `first next` $\cdots$ `next`

| $(n, b)$ | $t = 10$ | $t = 40$ | $t = 70$ | $t = 100$ |
|---|---|---|---|---|
| $(20, 5)$ | 0.64/49.68 | 4.04/– | 9.21/– | 14.84/– |
| $(30, 6)$ | 1.71/178.68 | 16.33/– | 36.23/– | 56.76/– |
| $(40, 7)$ | 4.01/454.98 | 38.55/– | 95.19/– | 152.79/– |
| $(50, 8)$ | 9.07/– | 100.34/– | 236.58/– | 374.07/– |
| $(60, 9)$ | 17.31/– | 183.89/– | 461.51/– | –/– |
| $(70, 10)$ | 32.25/– | 371.57/– | –/– | –/– |

(b) CP approach

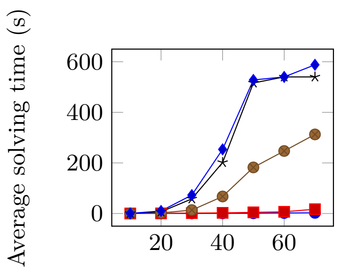
| $(n, b)$ | $t = 10$ | $t = 40$ | $t = 70$ | $t = 100$ |
|---|---|---|---|---|
| $(20, 5)$ | 0.663 | 0.435 | 0.562 | 1.075 |
| $(30, 6)$ | 0.435 | 0.560 | 0.780 | 1.011 |
| $(40, 7)$ | 0.562 | 0.519 | 0.799 | 1.139 |
| $(50, 8)$ | 0.762 | 0.521 | 0.767 | 1.102 |
| $(60, 9)$ | 1.002 | 0.501 | 0.835 | 0.975 |
| $(70, 10)$ | 1.425 | 0.526 | 0.873 | 0.1109 |

Table 2(a) shows the experimental results comparing the `@` operator against `first next` $\cdots$ `next`. Each table entry again shows the solving times using the new and old approaches respectively, separated by a "/", with "–" denoting a timeout. The results demonstrate our implementation significantly outperforming the previous approach, with up to 2 orders of magnitude speedup.

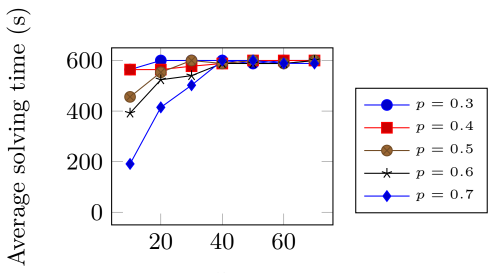
For the reader's reference, we also include Table 2(b), that is the solving time of Gecode finding a single solution/plan for the time-bounded scenario. Since St-CSP solvers find *all* solutions, it is reasonable to not be competitive with a traditional CP approach. However, when we asked for *all* solutions instead, **Gecode timed out** *for all but the* $t = 10$ *instances*, since the St-CSP search algorithm is able to avoid repeating equivalent search, via dominance detection. Asking a St-CSP solver to decide only the *existence* of *some* solution, instead of solving for all solutions, is scope for future work.

### 5.2 Path Planning in Grid World

The second set of experiments uses the path finding problem defined by the St-CSP model presented in Fig. 1. We generate random grid worlds of size $n \times n$ by independently sampling each directed edge between adjacent cells with probability $p$, as well as uniformly sampling the start and end points on the



(a) St-CSP approach: “until”



(b) Apt and Brand approach

Fig. 2: Path Planning: Eventuality condition

grid. Similarly, we performed two sets of experiments, solving for plans that eventually reach the goal (using the “until” constraint), and plans that have to reach the goal within a certain number of steps (using the `@` operator).

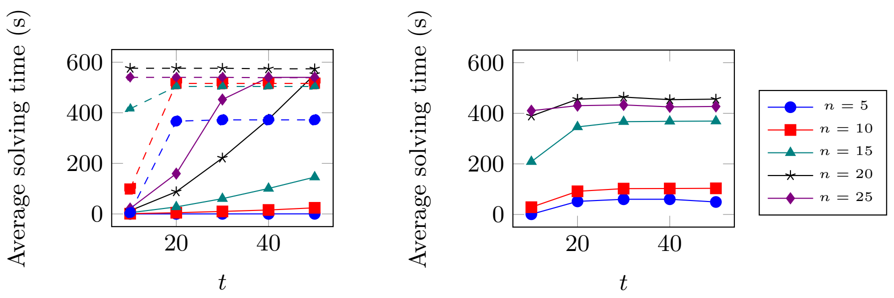
For the “until” experiments, we varied both $n$ and $p$, sampling 50 random instances for each setting of $n$ and $p$. Fig. 2(a) shows the average solving time of the test instances, where instances that timed out count as 600s. The solving times in this setting increase in $n$ polynomially, and become concave for larger $n$ and $p$ when a substantial number of instances start timing out.

For comparison, Fig. 2(b) shows the solving time using the Apt and Brand [1] framework. The figures show that most of the instances timed out, demonstrating that the St-CSP approach is far more efficient. Since any simple path on the grid has an upper bound of $n^2$ in length, similarly to the previous setting we also used a CP solver to solve for a plan of length $n^2$. However, Gecode runs into memory issues around $n = 40$, exceeding the 256GB memory available. Even before so, for $n = 10$ a significant proportion of the instances already timed out, even though the St-CSP solves them almost instantaneously (as in Fig. 2(a)). Because of the memory issues that Gecode ran into, we decided to not give corresponding runtime plots since runtime is ill-defined.

For our last set of experiments, we again replace the “until” constraint with the constraint that the path must have visited the end point by $t$ steps, a parameter that we vary across test cases. We generated 50 random instances for a selected set of $n$ values, however fixing $p = 0.8$ to make sure that a sizeable portion of the instances are satisfiable. We further varied $t$ on these instances.

Fig. 3(a) shows the average solving times by the old and new St-CSP approaches. We observe a 2 orders of magnitude improvement in solving time for large $t$. The plots for the `@` operator are also in general better behaved. We further found that the reason for the essentially horizontal plots for the “`first next` $\cdots$ `next`” operator phrase is due to it only being able to solve the trivially unsatisfiable instances in under 1 second, where the reachable component from the start point is small. All the other cases timed out, giving the plateau we observe in solving time for the operator phrase.

Fig. 3(b) shows the solving time using Gecode. The plots display similar plateauing behaviour as our old appproach, only starting earlier at $t = 20$.



(a) `@` (solid)/`first next` (dashed)    (b) CP approach

Fig. 3: Path planning: Time bounded

In comparison, the St-CSP approach is competitive with Gecode, despite the St-CSP solver being a prototype. We believe that it is due to the inherent specification complexity of the path planning problem on the grid. The entire graph structure has to be encoded for each time point, meaning that for the CP approach, the program is of size $O(tn^2)$, whereas the St-CSP is only of size $O(n^2)$.

## 6 Concluding Remarks

Our work improves the expressiveness of the St-CSP framework by augmenting it with 1) the new "until" constraint construct, adapted from the corresponding LTL operator, and 2) the `@` operator, which is a syntactic sugar for `first next` $\cdots$ `next` that further allows for faster solving by exploiting the special structure of the expression. We give corresponding new St-CSP solving algorithms, and also experimental evidence for their competitiveness with the corresponding CP approaches using Gecode. In our opinion the @ operator and the "until" constraint are for different purposes. The former is for time bounded scenario, while the latter is useful, for example, from a security perspective: we wish to know that our adversary can never achieve a sinister goal regardless of time budget.

By introducing the "until" constraint, we altered the structure of the generated solution automata and the guarantee we give regarding the execution of the automata (Section 3.3). From the statement that every run of the automaton is an accepting run, we weaken the guarantee (*whilst maintaining practical relevance*) to such that every finite run of the automaton could be extended to an infinite length solution stream. A natural direction for further investigation is to consider, under this weaker guarantee, how much more expressive can the St-CSP framework become. Are there other practical and natural constraints or temporal operators that, despite being currently inexpressible in the St-CSP framework, can be introduced with a solving algorithm that provides the above guarantee? Can we identify even weaker, yet still practically relevant guarantees that allows for even more expressiveness in the framework? We leave the answering of these questions for future work.

## References

1. Apt, K.R., Brand, S.: Infinite qualitative simulations by means of constraint programming. In: Proc. CP'06. pp. 29–43 (2006)
2. van Beek, P., Chen, X.: Cplan: A constraint programming approach to planning. In: Proc. AAAI'99/IAAI'99. pp. 585–590 (1999)
3. Büchi, J.R.: On a decision method in restricted second order arithmetic. In: Mac Lane, S., Siefkes, D. (eds.) The Collected Works of J. Richard Büchi, pp. 425–435. Springer New York (1990)
4. Dechter, R.: Constraint Processing. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2003)
5. Emerson, E.A.: Handbook of theoretical computer science (vol. b). chap. Temporal and Modal Logic, pp. 995–1072. MIT Press, Cambridge, MA, USA (1990)
6. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc. (2004)
7. Golden, K., Pang, W.: Constraint reasoning over strings. In: Proc. CP'03. pp. 377–391 (2003)
8. Harabor, D., Grastien, A.: Online graph pruning for pathfinding on grid maps. In: Proc. AAAI'11. pp. 1114–1119 (2011)
9. Kautz, H., Selman, B.: Planning as satisfiability. In: Proc. ECAI'92. pp. 359–363 (1992)
10. Kilby, P., Prosser, P., Shaw, P.: A comparison of traditional and constraint-based heuristic methods on vehicle routing problems with side constraints. Constraints 5(4), 389–414 (2000)
11. Lallouet, A., Law, Y.C., Lee, J.H.M., Siu, C.F.K.: Constraint programming on infinite data streams. In: Proc. IJCAI'11. pp. 597–604 (2011)
12. Lee, J.C.H., Lee, J.H.M.: Towards practical infinite stream constraint programming: Applications and implementation. In: Proc. CP'14. pp. 449–464 (2014)
13. Pesant, G., Gendreau, M., Potvin, J.Y., Rousseau, J.M.: An exact constraint logic programming algorithm for the traveling salesman problem with time windows. Transp. Sci. 32(1), 12–29 (1998)
14. Pnueli, A.: The temporal logic of programs. In: Proc. FOCS'77. pp. 46–57 (1977)
15. Sturtevant, N.R.: Benchmarks for grid-based pathfinding. IEEE Trans. Comput. Intell. AI in Games 4(2), 144–148 (2012)
16. Wadge, W.W., Ashcroft, E.A.: LUCID, the Dataflow Programming Language. Academic Press Professional, Inc. (1985)
17. Winskel, G.: The Formal Semantics of Programming Languages : An Introduction. MIT Press, Cambridge, MA, USA (1993)