Towards Practical Infinite Stream Constraint Programming: Applications and Implementation

Jasper C.H. Lee[1] and Jimmy H.M. Lee[2]

[1] Computer Laboratory, University of Cambridge, United Kingdom
chjl2@cam.ac.uk
[2] Department of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong
jlee@cse.cuhk.edu.hk

Abstract. Siu et al. propose stream CSPs (St-CSPs) as a generalisation of finite domain CSPs to cater for constraints on infinite streams, and a solving algorithm that produces a deterministic Büchi automaton recognising the solution language. As a novel application, we demonstrate how St-CSPs can model mathematically and generate a PID controller for driving a self-balancing tray and an inverted pendulum in real-time. We propose and prove the correctness of an improvement to the implementation that eliminates numerous unnecessary states in the solution automaton for St-CSPs involving the `first` temporal operator, thereby reducing solving time. We give two St-CSP examples that can benefit from our new implementation techniques. Our approach always generates a solution automaton not bigger than, but potentially exponentially smaller than, that produced by the original implementation. Experimental results show substantial improvements.

1 Introduction

Streams of data are ubiquitous. They can either be discrete sequences on their own (e.g. stock market data), or discrete samples of continuous signals (e.g. positional data with respect to time). The evolution of such sequences is typically governed by some physical laws or mathematical equations. However, standard finite domain constraint satisfaction problems (CSPs) do not model such problems very well, because they can only model a finite segment of an otherwise infinite problem. To model such discrete time constraint problems more naturally, Siu et al. [8, 11] introduce stream constraint satisfaction problems (St-CSPs) by adapting temporal operators in Wadge and Ashcroft's [12] Lucid programming language. They give the definition of St-CSPs, and a solving algorithm that produces a deterministic Büchi automaton recognising the solutions of an St-CSP. The termination, soundness and completeness of their algorithm are proven. They also suggest practical applications for St-CSPs, such as generating harmonic accompaniment to a melody, and the game engine for the once popular Digi Invaders[3] game in early Casio calculators in the 1970s.

This paper is about practical stream constraint programming. The goal is to push the limit of this relatively new member of the CP family, and take the first step towards

[3] http://www.youtube.com/watch?v=1YafgAcmov4

putting the theoretical framework into practice. Since there are currently no common modelling idioms, and we know little about implementation technology and applications, we approach this idea from two angles. First, we demonstrate that St-CSPs can be used for solving interesting practical real life problems. Continuing the work on game engine generation [8, 11], we model real-time hardware controllers as St-CSPs. Even though discretisation and approximations have to be applied, we find that the approach produces stable control on our hardware. Second, we propose an improvement for the search algorithm to reduce solving time and the size of the solution automaton. Our improvement is restricted to a certain class of St-CSPs, and we give practical usages of this class on two applications. We also prove the correctness of our technique. To demonstrate the efficiency of our proposal, we give experimental results to compare our new search algorithm against the original, showing orders of magnitude improvement both in terms of runtime and solution automaton size.

2 Background

This section introduces the technical background for stream constraint solving. We first state the definition of St-CSPs and related notions, followed by the constraint specification language. The solving algorithm of Siu et al. [8, 11] is summarised.

2.1 Infinite Strings and Stream Constraint Satisfaction Problems

An *infinite string* α over an *alphabet* Σ is a function $\mathbb{N}_0 \to \Sigma$. Given i, $\alpha(i)$ is an individual *daton* of α at time point i. The set of all such strings with alphabet Σ is denoted Σ^ω. Infinite strings are also referred to as *streams*.

The notation $\alpha' = \alpha(i, \infty)$ is used for the *string suffix* $\alpha'(j) = \alpha(j + i)$. For a language L, $L(i, \infty) = \{\alpha(i, \infty) \mid \alpha \in L\}$. As for a *finite prefix* of a string, infinite or not, the notation $\alpha' = \alpha[0 : i]$ is used to denote the string $\alpha'(j) = \alpha(j)$ if $0 \leq j \leq i$ and undefined otherwise. A special case is when $i < 0$, denoting the empty string.

A *stream constraint satisfaction problem* (St-CSP) is a tuple $P = (X, D, C)$ [8, 11], where $X = \{x_1, \dots, x_n\}$ is a finite set of *variables*, $D(x) = (\Sigma(x))^\omega$ is a function that maps a variable to its *domain* which is the set of all infinite strings with alphabet $\Sigma(x)$, C is a finite set of *constraints*. A constraint $c \in C$ is a relation R defined on an ordered subset $Scope(c)$ of variables. The relation gives all the valid simultaneous assignments of values to variables in $Scope(c)$. Every constraint $c \in C$ must also be a deterministic ω-regular language [2].

An *assignment* $A(x_i) \in D(x_i)$ is a function mapping a variable to an element in its domain. A *satisfies* a constraint c if and only if $(A(x_{i_1}), A(x_{i_2}), \dots, A(x_{i_k})) \in c$, where $Scope(c) = (x_{i_1}, \dots, x_{i_k})$. The notion can be generalised to say that the string β of tuples $\beta(i) = (A(x_1)(i), \dots, A(x_n)(i))$ *satisfies* the constraint c where $X = \{x_1, \dots, x_n\}$. An St-CSP is *satisfied* by a variable assignment A or a string of tuples β if and only if all constraints are satisfied.

As a corollary of the closure properties of deterministic ω-regular languages, the solution set $sol(P) = \{t = (a_1, a_2, \dots, a_n) \in \prod_i D(x_i) \mid \forall\, c \in C.\, t \text{ satisfies } c\}$ of an St-CSP P is also a deterministic ω-regular language.

In addition, two St-CSPs P and P' are said to be *equivalent*, written as $P \equiv P'$ as usual, if and only if $sol(P) = sol(P')$.

Given a set of constraints C and a point i, the *shifted view* (previously known in the literature as the *limited view* [8, 11]) of C is defined as $C(i, \infty) = \{c_k(i, \infty) \mid c_k \in C\}$. Similarly, given an St-CSP $P = (X, D, C)$ and a point i, the *shifted view* of P is defined as $\hat{P}(i) = (X, D, C(i, \infty))$.

2.2 The Stream Constraint Language

In this paper, we are only concerned with St-CSPs whose variable alphabets are integer intervals, i.e. $[m, n]^\omega$ for some $m \leq n \in \mathbb{Z}$.

To specify constraints, 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 a where $a(n) = 2$.

Three *temporal* operators, in the style of the Lucid programming language [12], `first`, `next` and `fby`, are defined on streams. Suppose α and β are streams. We have `first` α being the constant stream of $\alpha(0)$, and `next` α being the "tail" of α, i.e. `next` $\alpha = \alpha(1, \infty)$. In addition, α `fby` $\beta = \gamma$ is the concatenation of the head of α and β, i.e. $\gamma(0) = \alpha(0)$ and $\gamma(i) = \beta(i-1)$ for $i \geq 1$.

Furthermore, there are *pointwise* operators, such as integer arithmetic operators {`+`, `-`, `*`, `/`, `%`}. They combine two streams point by point using the corresponding arithmetic operator. Integer arithmetic relational operators are {`lt`, `le`, `eq`, `ge`, `gt`, `ne`}. They compare the two argument streams pointwisely and return a *pseudo-Boolean stream*, that is a stream in $[0, 1]^\omega$, where 0 denotes $false$ and 1 denotes $true$. Pointwise Boolean operators {`and`, `or`} act on any two pseudo-Boolean streams γ and η. The final pointwise operator supported is `if-then-else`. Suppose γ is pseudo-Boolean, and α, β are streams in general, then (`if` γ `then` α `else` β)(i) is $\alpha(i)$ if $\gamma(i) = 1$ and $\beta(i)$ if $\gamma(i) = 0$.

Given these stream operators, we can now use the following relations to express stream constraints. For integer arithmetic comparisons $\circ \in \{<, <=, ==, >=, >, !=\}$, the constraint $\alpha \circ \beta$ is *satisfied* if and only if the arithmetic comparison $\circ$ 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.

Care should be taken to distinguish between constraints and the relational operators. Relational operators take two streams and gives a pseudo-Boolean stream as an output. Constraints, on the other hand, are relations on streams.

2.3 Normalising Constraints

Siu [11] defines an St-CSP to be in *normal form* if it contains only *primitive constraints*. Primitive constraints are in one of the following three forms, assuming x_i are stream variables.

- Primitive first constraints: `first` x_i == `first` x_j
- Primitive next constraints: x_i == `next` x_j
- Primitive pointwise constraints: Constraints not involving `first`, `next` or `fby`.

Reducing all occurrences of `first` operators to the primitive form is beneficial, since primitive first constraints can be enforced like a primitive pointwise constraint, but can be deleted after the first time point.

All St-CSPs are reduced to an equivalent normal form before being submitted to the solver. Siu [11] also gives a simple recursive translation of an St-CSP into this normal form. Only the appearance of either "`first` *expr*", "`next` *expr*" or "$expr_1$ `fby` $expr_2$" may violate the normal form property. The cases are translated separately. By adopting notations from programming language semantics theory [13], we write $c\,[–]$ for *constraint contexts*, i.e. constraints with placeholders for syntactic substitution. For example, if $c\,[–]$ = [– + 3 >= 4], then $c\,[\texttt{first}\ \alpha]$ = [($\texttt{first}\ \alpha$) + 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 has to be further normalised, and C_1 is the set that is guaranteed to be in normal form already. Hence, the initial constraint pair for the St-CSP (X, D, C) is $(C, \{\})$, and the rules are applied until none are applicable.

1. $(C_0 \cup \{c\,[\texttt{first}\ expr]\}, C_1) \rightsquigarrow (C_0 \cup \{c\,[v_1], v_2 == expr\}, C_1 \cup \{\texttt{first}\ v_1 == \texttt{first}\ v_2, v_1 == \texttt{next}\ v_1\})$ where v_1 and v_2 are auxiliary variables not in any of $c\,[–]$, C_0 and C_1.
2. $(C_0 \cup \{c\,[\texttt{next}\ \ expr]\}, C_1) \rightsquigarrow (C_0 \cup \{c\,[v_1], v_2 == expr\}, C_1 \cup \{v_1 == \texttt{next}\ v_2\})$ where v_1 and v_2 are auxiliary variables not in any of $c\,[–]$, C_0 and C_1.
3. $(C_0 \cup \{c\,[expr_1\ \texttt{fby}\ expr_2]\}, C_1) \rightsquigarrow (C_0 \cup \{c\,[v_1], v_2 == expr_1, v_3 == expr_2\}, C_1 \cup \{\texttt{first}\ v_1 == \texttt{first}\ v_2, \texttt{next}\ v_1 == v_3\})$ where v_1, v_2 and v_3 are auxiliary variables not in any of $c\,[–]$, C_0 and C_1.

2.4 Search Trees

A *search tree* for an St-CSP P is a tree with potentially infinite height. Its nodes are St-CSPs, and the root node is P itself. The *level* of a node N is recursively defined as 0 for the root node, and $1 + \ell$ for non-root nodes N where ℓ is the level of the parent of N. A child node $Q' = (X, D, C \cup \{c'\})$ is constructed from a parent node $P' = (X, D, C)$ at level k and an *instantaneous assignment* $\tau(x) \in \Sigma(x)$, where τ takes a stream variable x and returns a daton value for it. In other words, τ gives a scalar assignment to the daton of streams 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 τ. Fig. 1 shows an example search tree.

It is also useful to identify a search node with its shifted view. For a search node Q at level k, we identify it 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 pointwise 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 $\{\tau_i\}$ is the sequence of instantaneous assignments that constructs F from the root node, i.e. $P \xrightarrow{\tau_0} \ldots \xrightarrow{\tau_k} F$. We say node F is a *failure* if and only if $\{\tau_i\}$ violates a constraint $c \in C$.

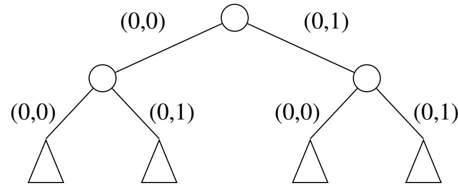

Fig. 1: Example Search Tree

2.5 Solving St-CSPs

Given an St-CSP P, its search tree is explored using depth first search. Backtracking happens when the current search node is a failure. A search node A at level k *dominates* [8, 11] another search node B at level k', written as $B \prec A$, if and only if their shifted views are equivalent ($\hat{A}(k) \equiv \hat{B}(k')$) and A is visited before B during the search. When the algorithm visits a search node N that is dominated by a previously seen node M, the edge pointing to N is redirected to M instead. The resulting structure is isomorphic to a deterministic Büchi automaton, which accepts all and only the solutions of P. Siu et al. [8, 11] prove the termination, soundness and completeness of the algorithm.

3 Application on Real-Time PID Control

A *proportional-integral-derivative* (*PID*) controller [10] is a loop feedback control mechanism. A process receives an input signal $u(t)$ and gives an output $y(t)$, which has an error $e(t) = y(t) - r(t)$ from a reference signal $r(t)$. The PID controller produces the input signal $u(t)$ by adding a weighted sum of the following three components. The proportional component is simply the error signal $e(t)$. The integral component is $\int_0^t e(\tau)\, d\tau$. The derivative component is $e'(t)$. Fig. 2 shows the described structure. K_p, K_i and K_d are the corresponding coefficients for the components in the weighted sum.

We can model a PID controller as an St-CSP. The first step is to discretise and scale the domain of the error signal, such that the signal can be represented as an integer stream e. For example, the error signal might have a real interval $[-15, 15]$ as the domain representing an angle deviation. A possible discretisation is to map the interval to the integer interval $[-60, 60]$ by multiplication with 4 and rounding. The stream e (the error and the proportional component) is unconstrained and acts as an input to the automaton. At each state, the edge with the correct error value is selected in order to proceed to the next state.

There is a tradeoff between having greater precision in the error stream and limiting the size of the solution automaton. The standard approaches to determining the PID coefficients are by experimentation or analysis of a mathematical model of the process.

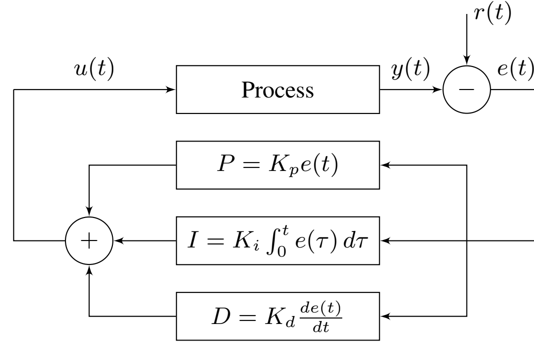

Fig. 2: PID Controller Schematics

A good discretisation of the error stream can therefore be similarly determined using either of the approaches.

Given the discretisation, we can model the derivative component of the PID control signal. For a stream of discrete time signals α, an analog of the derivative is the finite difference $\alpha(i+1) - \alpha(i)$. Any linear scaling factor required for a better approximation can simply be absorbed into the K_d coefficient for the weighted sum. In order to compute finite differences, a stream l is introduced with constraints l == 0 `fby` e. The derivative stream d is therefore constrained by d == e − l. From this, we deduce that the bounds for d is $[-2c, 2c]$ if the bounds for e is $[-c, c]$.

An analog of the integral for discrete signals is the finite sum $\sum_0^n \alpha(i)$. In an ideal PID controller, the integral component can be unbounded. In practice, it is either restricted by the real number representation of the machine or artificial bounds are introduced. In our case, a bound b is also needed in order to have a finite alphabet for the integral stream i. With the value of b decided, the integral stream can be computed using the constraint i == 0 `fby` (`if` (i+e `gt` b) `then` b `else` (`if` (i+e `lt` −b) `then` −b `else` i+e)). There is an alternative approximation for the integral stream if we know the scaling factor applied to it is close to 0. In this case, instead of summing the error discrete signals, we sum the sign of the error signals. That is, we introduce another stream $tempI$ with constraint $tempI$ == 0 `fby` (i + `if` (e `gt` 0) `then` 1 `else` (`if` (e `lt` 0) `then` −1 `else` 0)). Instead of using the previous constraint for computing i, we use i == `if` ($tempI$ `gt` b) `then` b `else` (`if` ($tempI$ `lt` −b) `then` −b `else` $tempI$). It is also possible to inline the definition of $tempI$ into i, but we present this as it is here for clarity. This alternative approximation is useful for keeping the alphabet of the stream i small. Fig. 3 gives a generic PID controller model.

As discrete time controllers process input and output streams, St-CSPs are ideal for modelling them. The typical way of implementing controllers is to program the controller equations in an imperative language. Programming with destructive assignments and various control flow commands can be error prone. Bentley [1] gives experimental results that only 10% of professional programmers write correct code for an algorithm as simple as the binary search. Using the St-CSP approach, the imperative code required in a program is only for traversing a solution automaton according to the sensor error input stream and producing control signals to the output streams. This code has to be engineered only once and is largely reusable. The St-CSP specification language is declarative in nature without any side effects in its semantics. Hence, it inherits the

Streams e, ℓ with alphabet $[-c, c]$
Stream i with alphabet $[-b, b]$
Stream d with alphabet $[-2c, 2c]$

d == $e - \ell$
ℓ == 0 `fby` e
i == 0 `fby` (`if` (i+e `gt` b) `then` b
        `else` (`if` (i+e `lt` $-b$) `then` $-b$ `else` i+e))

Fig. 3: Generic PID Controller

advantages of declarative programming over imperative programming, including readability, conciseness, compositionality and referential transparency. Correctness and elegance are therefore more easily achievable than using a conventional programming language like C.

A PID controller for a self-balancing tray[4] was synthesised. The platform has a tray holding a pingpong ball, two motors that allows it to rotate in 3D space and an accelerometer that measures the orientation. The purpose of the controller is to maintain the horizontal position of the tray as the platform rotates, such that the pingpong ball does not fall out. We also applied the technique to control a self-balancing inverted pendulum[5]. It has a vertical body, with wheels at the bottom to allow movement for balancing the body as it tilts sideways. The controller actually uses a variant of PID control with a second derivative component in addition to the original three. Also, a complementary filter and a Kalman filter were applied to the gyroscope sensor input to eliminate noise. The filters however are not part of our St-CSP model.

The traditional controllers of the above hardware happen to be simple and small, even when implemented in C. We anticipate the advantages of our approach to become more apparent when the controllers are more complex. The purpose of the current exercise is really to demonstrate that CP can have applications in real-time hardware control.

4 Improved Handling of the `first` Operator

Our new approach focuses on the handling of streams constructed using the `first` operator. Fig. 4 contains two St-CSPs that show some uses of `first` would increase the number of states in the solution automaton, and some other uses would not. Problem 1 imposes that the first daton of x has to be less than the first daton of y, whilst Problem 2 requires all datons of x to be less than the first daton of y. Therefore, the constraint in Problem 1 only concerns the first time point, whereas the effect of the constraint in Problem 2 persists indefinitely.

The optimal solution automaton (Fig. 5), in the sense of having the fewest states, for the St-CSP in Fig. 4a has two states, whilst the optimal solution automaton (Fig. 6)

[4] A video demonstration of the self-balancing tray in operation can be found at `http://www.youtube.com/watch?v=dT56qAZt8hI`

[5] A video demonstration of the inverted pendulum can be found at `http://www.youtube.com/watch?v=5GvbG3pN0vY`

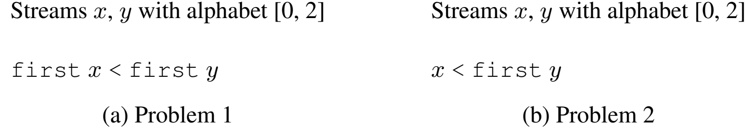

Fig. 4: Example St-CSPs

for the St-CSP in Fig. 4b is a three-state automaton. In fact, for the St-CSP in Fig. 4b, as the size of the alphabet of y increases, the number of states in the optimal solution scales linearly. This is because, for each value that `first` y takes, there is a different upper bound on x. Therefore, a different state is needed for each value of `first` y.

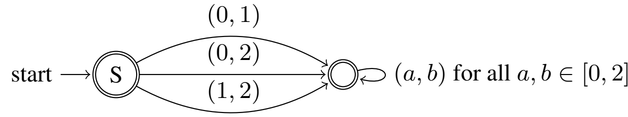

Fig. 5: Optimal Solution Automaton for `first` x < `first` y

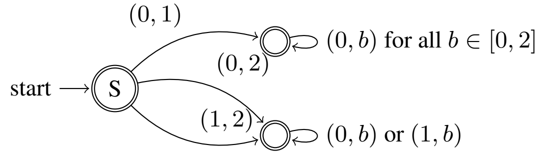

Fig. 6: Optimal Solution Automaton for x < `first` y

The difference between the two St-CSPs is that Problem 1 has a constraint that involves only the first time point, whereas Problem 2 has a constraint that involves streams with `first` operators and also other constructions of variable streams. These two examples demonstrate that constraints of the former kind do not increase the solution automaton size in general, whilst constraints of the latter kind can potentially multiply the size by a linear factor in the size of the stream alphabet.

However, the original solving approach [8, 11] produces an automaton (Fig. 7) of linear size even for Problem 1, as a result of their normalisation rules. Stream expressions of the form "`first` *expr*" are normalised with the introduction of primitive next constraints (x_i `==` `next` x_j), which increase the size of the solution automaton because the daton values taken by x_i has to be taken by the daton of x_j at the next time point. Therefore, different states are needed to distinguish between the different values, effectively acting as memory for the automaton. Figure 8 shows an example of how states act as memory, where the alphabet of x_i is $[0, 1]$ for simplicity. Each state in Fig. 8 is annotated with the last daton value of the stream x_j that it represents.

Our proposed approach therefore is designed to avoid introduction of primitive next constraints for normalising streams with `first` operators, by improving the normalisation and search procedure. Even though the proposal applies only to a certain class of St-CSPs, we identify two practical uses for this class, which is presented in Sect. 5.

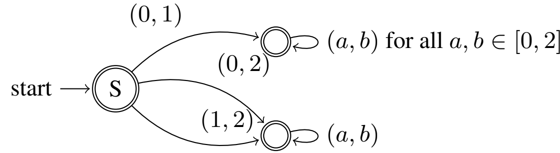

Fig. 7: Siu et al. [8, 11]: Solution Automaton for `first` x < `first` y

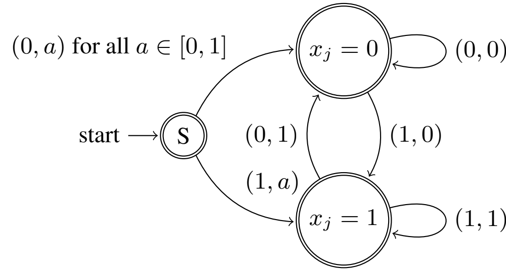

Fig. 8: Solution Automaton for x_i == `next` x_j

4.1 Constraint Normalisation

We propose to relax Siu's normal form [11]. An St-CSP is in *normal form* if it contains only constraints of the following two forms.

- Primitive next constraints: x_i == `next` x_j
- Primitive pointwise constraints that do not involve `next` or `fby`.

Note that, in our approach, constraints involving only the `first` temporal operator are also considered as pointwise constraints.

The normalisation of `next` and `fby` streams is largely unchanged from Siu's algorithm [11]. The following is our new normalisation algorithm concerning `first` streams.

1. $(C_0 \cup \{c[\texttt{next first}\ \mathit{expr}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[\texttt{first}\ \mathit{expr}]\}, C_1)$
2. $(C_0 \cup \{c[\texttt{first first}\ \mathit{expr}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[\texttt{first}\ \mathit{expr}]\}, C_1)$
3. $(C_0 \cup \{c[\texttt{first}\ (\mathit{expr}_1\ \texttt{fby}\ \mathit{expr}_2)]\}, C_1) \rightsquigarrow (C_0 \cup \{c[\texttt{first}\ \mathit{expr}_1]\}, C_1)$
4. $(C_0 \cup \{c[\texttt{first}\ \mathit{const}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[\mathit{const}]\}, C_1)$ where *const* is a constant stream.
5. $(C_0 \cup \{c[\texttt{first next}\ \mathit{expr}]\}, C_1) \rightsquigarrow (C_0 \cup \{c[\texttt{first}\ v]\}, C_1 \cup \{v == \texttt{next}\ \mathit{expr}\})$ where expr is not of the form `first` expr_1, `next` expr_1 or expr_1 `fby` expr_2, and v is an auxiliary variable not in any of $c[–]$, C_0 and C_1.

To calculate the alphabets of auxiliary variables, we use interval arithmetic to construct bounds of the expression represented by the variable.

Given our new definition of normal form, the search algorithm has to be adapted.

4.2 Search Algorithm

Constraint specifications are now assumed to be in the normal form defined in the last section. The search algorithm we propose is again similar to the original approach [8, 11], but the constraints in our approach can change during search.

We now describe how we construct the set of constraints C' of a child node $\hat{Q}(k+1)$ from the set of constraints C of the parent node $\hat{P}(k)$ and instantaneous assignment τ_k if $\hat{P}(k) \stackrel{\tau_k}{\to} \hat{Q}(k+1)$. The construction of τ_k should have been such that it satisfies all the primitive next constraints imposed by τ_{k-1}, i.e. $\tau_k(x_j) = \tau_{k-1}(x_i)$ for all primitive next constraints x_i == `next` x_j, and also all primitive pointwise constraints in C. In order to construct C', a direct copying from C is not correct. We observe that primitive next constraints are invariant in all shifted views of an St-CSP. It is only the primitive pointwise constraints that may change. In particular, streams with `first` operators are no longer the same when we take the shifted view of the child node $\hat{Q}(k+1)$. Such streams have to be *evaluated*, meaning that all variable streams inside a `first` operator have to be substituted by their assigned values from τ_k. The resulting stream expressions thus contain only constant streams, pointwise operators and `first` operators. Since `first` operators have no effect on constant streams, the expressions can always be reduced to a constant stream by evaluating pointwise operations and deleting `first` operators. We say we *evaluate* a constraint if and only if we evaluate all the stream expressions with `first` operators in the constraint. After evaluating a constraint with τ_k, we test whether it is a tautology. Since arithmetic is not decidable in general, we only consider tautologies of the zeroth order, i.e. those not involving universally quantified variables. If the constraint is a tautology, it is removed from the constraint set C.

Example 1. An example is given here to illustrate the construction process. Consider the St-CSP in Fig. 9a, which is normalised to the one in Fig. 9b. We construct the child node $\hat{Q}(1)$ from the root node and the instantaneous assignment $\tau_0 = \{(x = 0), (y = 0), (v = 2)\}$. Observe that given this τ_0, any τ_1 must obey $\tau_1(y) = 2$ due to the primitive next constraint. The final result of C' is $\{x$ == 0, x < v, v == `next` y$\}$:

- `first` x == `first` (x + y) is first substituted with values of τ_0 and becomes `first` 0 == `first` (0 + 0). The constraint is then evaluated into 0 == 0, which is a tautology not involving any variables. Therefore, we remove the constraint from C'.
- x == `first` y is evaluated into x == 0. Since x is still present, the constraint is not removed from C'.
- v == `next` y and x < v are unchanged.

Dominance between search nodes is detected in the same way as the original approach [8, 11]. We say x is a *signature stream* if and only if x appears on the L.H.S. of a primitive next constraint x == `next` y for some stream y. Suppose $\hat{Q}_1(k_1)$ is a search node constructed from the instantaneous assignment τ_{k_1} and $\hat{Q}_2(k_2)$ is another search node constructed from τ_{k_2}. Let their corresponding sets of constraints be C_1 and C_2. We say that the two search nodes are *equivalent* if and only if C_1 is syntactically equivalent to C_2 and $\tau_{k_1}(x) = \tau_{k_2}(x)$ for all signature streams x. The proofs of Siu et al. [8, 11] can be easily adapted to show that the detection is sound.

Streams x, y with alphabet [0, 2]

```
first x == first (x + y)
x == first y
x < next y
```

(a) Example St-CSP

Streams x, y, v with alphabet [0, 2]

```
first x == first (x + y)
x == first y
x < v, v == next y
```

(b) Normalised Example St-CSP

Fig. 9: Example St-CSPs Illustrating the Search Algorithm

The previous example also demonstrates how we remove all information about the values taken for constraints involving only the first time point. They are always evaluated into a tautology that we recognise. Therefore, it is impossible for the solution automaton to have any "memory" on what values were taken, meaning there are no distinct states distinguishing between the different values. This achieves the reduction in automaton size we seek.

Our improvement is applicable whenever there exists a stream expression of the form `first` *expr* that only appears in constraints involving the first time point, and can take multiple values. The size of the solution automaton can be reduced by an exponential factor from the original approach [8, 11]. For example, for an St-CSP with $2n$ streams with alphabet $[0, 1]$ and constraints `first` x_{2i} `==` `first` x_{2i+1} for $0 \leq i \leq n-1$, our approach produces a two state automaton since all constraints are removed after the first time point. In contrast, the original approach [8, 11] normalises the problem and produces $2n$ streams constrained by primitive next constraints. There are a total of 2^n valid combinations of values taken by the streams with `first` operators. Therefore, considering also the start state, an automaton of size $2^n + 1$ is produced.

We prove the soundness of our constraint construction algorithm in the following. Recall from Sect. 2.4 that, if a parent node P has the set of constraints C, then the constraints of a child node Q are $(C \cup \{c'\})$ where c' is the constraint stating $x(0) = \tau_k(x)$ for all streams x and $x(i)$ is unconstrained for all $i > 0$. Since our construction algorithm takes shifted views into account, the following theorem proves the equivalence of C' and $(C \cup \{c'\})(1, \infty)$ where C' is the result of our construction.

Theorem 1. *Suppose the constraint set C' of the shifted view of child node $\hat{Q}(k+1)$ is constructed from the constraint set C of the parent node $\hat{P}(k)$ and the instantaneous assignment τ_k. Then $C' = (C \cup \{c'\})(1, \infty)$ where c' is the constraint stating $x(0) = \tau_k(x)$ for all streams x and $x(i)$ is unconstrained for all $i > 0$.*

Proof. $C' \subseteq (C \cup \{c'\})(1, \infty)$: Let $a(1, \infty)$ be a string satisfying C' and $a(0) = (\tau_k(x_1), \ldots, \tau_k(x_n))$ where $\{x_i\}$ is the set of variables. Therefore a satisfies c' by construction. Consider an arbitrary constraint $c \in C$. If c is a primitive next constraint, then a must satisfy c as $c \in C'$ by construction. Otherwise, c is a primitive pointwise constraint. If c does not involve `first` operators, then $a(1, \infty)$ must also satisfy c as $c \in C'$ by construction. If c does involve `first` operators, then $a(1, \infty)$ satisfies the evaluated version of c, for the evaluated constraint is in C'. Since stream expressions with `first` operators are evaluated numerically according to $\tau_k = a(0)$, $a(1, \infty)$ must therefore also satisfy c. The pointwise interpretation of c is satisfied by $a(0) = \tau_k$ by construction, and therefore a satisfies c as a whole. To summarise, a satisfies any

constraint $c \in C$ and also the constraint c'. Hence $a(1,\infty)$ satisfies $(C \cup \{c'\})(1,\infty)$, proving $C' \subseteq (C \cup \{c'\})(1,\infty)$.

$(C \cup \{c'\})(1,\infty) \subseteq C'$: Let $b(1,\infty)$ be a string that satisfies $(C \cup \{c'\})(1,\infty)$ and $b(0) = \tau_k$. Stream b therefore satisfies $(C \cup \{c'\})$ by construction. Consider an arbitrary constraint $c \in C'$. If c is a primitive next constraint, then $c \in C$ as well, and hence $b(1,\infty)$ satisfies c. Otherwise, c is a primitive pointwise constraint. If $c \in C$, then $b(1,\infty)$ satisfies c. If not, then c is evaluated from a constraint $c_0 \in C$ using values of τ_k. By definition, b satisfies c_0. Therefore, b satisfies the constraint d constructed by evaluating streams in c_0 using values of τ_k. Stream d is thus a constraint not involving any `first` operators, which implies that $b(1,\infty)$ satisfies d. Also observe that d is in fact c, and so b satisfies c. Since c does not involve any temporal operators, $b(1,\infty)$ must also satisfy c. From the above, regardless of the type of constraint c is, $b(1,\infty)$ satisfies c. Therefore, $(C \cup \{c'\})(1,\infty) \subseteq C'$. □

5 Benefitting from the New Implementation

Our improvement applies only to certain usages of the `first` operator in an St-CSP. In this section, we show possible uses of the `first` operator in applications that can benefit from our new search algorithm.

5.1 Symmetry Breaking

The `first` operator can be used for breaking solution symmetry [4] in St-CSPs to reduce search, in the same way symmetry breaking helps with solving standard CSPs.

Symmetry breaking is the avoidance of visiting symmetric counterparts of visited search space. Suppose a CSP has *value symmetry* [9] σ and *variable symmetry* [9] σ'. If $\{x_0 = d_0, x_1 = d_1, \ldots, x_n = d_n\}$ is a solution of P, then $\{x_0 = \sigma(d_0), \ldots, x_n = \sigma(d_n)\}$ and $\{x_{\sigma'(0)} = d_0, \ldots, x_{\sigma'(n)} = d_n\}$ are also solutions respectively.

One technique for breaking value symmetry is by preassignment [7]. An analogous technique for stream constraint solving is to preassign the first daton of streams by constraints of the form "`first` x `==` *const*". This is a constraint that only concerns the first time point. However, since the stream with the `first` operator can only take one value, preassignment constraints do not increase the sizes of solution automata produced by even the original approach [8, 11].

To break variable symmetry, a lexicographical ordering of assigned values can be imposed [5]. That is, suppose there is a fixed ordering on the set of variables. Extra constraints are added to enforce that if $x_1 < x_2$, then $d_1 \leq d_2$ where d_1 and d_2 are the values assigned to x_1 and x_2 respectively. An analogous treatment with streams is to enforce such ordering at the first time point by adding constraints such as `first` x_1 `<=` `first` x_2 and `first` x_2 `<=` `first` x_3.

Observe that streams with `first` operators in these lexicographical ordering constraints can take multiple values in general. Therefore, our improvement applies and produces a smaller solution automaton than the original approach [8, 11]. Given n

streams with the same alphabet of size $|\Sigma|$, our solution automaton is smaller[6] by a multiplicative factor $\binom{|\Sigma|+n-1}{n}$, which is the number of different valid assignments for the first datons of each stream. Section 6 includes experimental results to show the improvement.

5.2 Sequential Planning

Ghallab et al. [6] give a framework for encoding planning problems in traditional CSPs. The framework first states a planning problem in the *state variable representation*, consisting of *state variables* which are descriptions of the world that can change over time, *actions* with *preconditions* and *effects* which cause changes to the world, and *rigid relations* which describe the invariants in the world. For example, `at(cat)` is a state variable that holds the location of the `cat` object. The `move(o,a,b)` action has the precondition that `at(o) = a` and the effect that `at(o) = b`. `adjacent` = $\{$`(desk,wall), (desk,bed)`$\}$ is a rigid relation.

The state variable representation can then be encoded [6] as a CSP that expresses a plan of length t. Each time point has an associated $\texttt{action}^t$ variable, denoting the action taken at time t. Each state variable is encoded as a constraint variable for every time point, for example $\texttt{at(cat)}^0, \ldots, \texttt{at(cat)}^3$ for a plan of length 3. Precondition constraints are used to enforce the preconditions of actions. They are of the form ($\texttt{action}^t$ = `a`) $\Rightarrow$ (preconditions of `a` at time $t-1$), such as ($\texttt{action}^t$ = `move(o,a,b)`) $\Rightarrow$ ($\texttt{at(o)}^{t-1}$ = `a`). Similarly, there are effect constraints of the form ($\texttt{action}^t$ = `a`) $\Rightarrow$ (effects of `a` at time t). Finally, frame constraints enforce that actions do not change anything other than their effects. For example, the constraint $\{(\texttt{action}^t = \texttt{move(o,a,b)}, \texttt{has(balloon)}^t = \texttt{c}, \texttt{has(balloon)}^{t+1} = \texttt{c} \mid \texttt{c}$ is an object$\}$.

With a St-CSP formulation, however, a variable for each time point is no longer needed. Only one St-CSP variable is required for each state variable, and another St-CSP variable for the action stream as St-CSP variables inherently span all time points. Precondition and effect constraints do not need to be specified per time point either. Observe that implication can be specified by inequality of pseudo-Boolean streams. Hence the precondition constraints ($\texttt{action}^t$ = `move(o,a,b)`) $\Rightarrow$ ($\texttt{at(o)}^{t-1}$ = `a`) can be specified in an St-CSP as a single constraint `next` ($action$ `eq` $move(o,a,b)$) `<=` $at(o)$ `eq` a. It is a coincidence that the inequality appears typographically in the reverse direction of the implication symbol.

Subsequently, to ensure that the goal is achieved within t time points, a goal constraint is added: `first next next next ... next` *goal* == 1 where there are t `next` operators and *goal* is the pseudo-Boolean stream expression denoting whether the goal has been achieved or not.

With traditional CSPs, the size of the specification for a t step planning problem scales linearly with t. With St-CSPs, the size stays constant. Therefore, the St-CSP approach achieves representational simplicity that is not possible with the standard CSP approach. Another advantage is that, even though there may be potentially infinitely

[6] This calculation excludes the start state, i.e. the size of the original automaton is $\binom{|\Sigma|+n-1}{n} \times (|S|-1)+1$ where $|S|$ is the size of our automaton.

many solutions to the St-CSP, the solution set can be represented by a finite description, namely the solution automaton.

Example 2. The following is a simple example demonstrating the modelling technique. Suppose there is a unique, physical document to be circulated to n individuals. Only one individual may hold the document at any single time point. We use the state variables `seen(i)` to denote whether individual `i` has seen the document yet. The action `giveTo(i)` has no preconditions and the effect that `seen(i)` becomes true (or 1). Using the formalism above, we get the simple St-CSP in Fig. 10. In this example, we do not specify the length of the plan.

Stream $goal$ with alphabet [0, 1]
Stream $giveTo$ with alphabet [0, $n-1$]
Streams $seen_0, \ldots, seen_{n-1}$ with alphabet [0, 1]

Constraints:
For each i,
`first` $seen_i$ == 0
`next` $seen_i$ >= $seen_i$ `or` ($giveTo$ `eq` i)

`first` $goal$ = 0
$goal$ >= ($seen_0$ `and` ... `and` $seen_{n-1}$)
`next` $goal$ >= $goal$;

Fig. 10: Document Circulation Planning St-CSP

We can also use the `first` operator to specify initial conditions. For example, it can be the case that the first individual who gets the document must be one of senior rank, which is defined by the individual having a number smaller than $(n-1)/2$. We introduce a constraint `first` $giveTo$ < $(n-1)/2$. Initial conditions produce constraints that only involve the first time point. When an initial condition is not strict, such as the one above, the streams with `first` operators can take multiple values. In this case, our improvement applies again, resulting in a smaller solution automaton and faster search. This shows that our new implementation has relevance to planning problems as well.

6 Experimental Results

We compare our implementation with the original implementation [8, 11] using both the runtime and the size of the solution automaton as metrics. Experiments are conducted on an Intel Core i7 (4 × 2.2GHz) machine with 16GB RAM. Both solvers are set to timeout in 1 hour. A "-" in the results table means the solver failed to solve the test case within the time limit. We also highlight the best results in bold per test case in the tables. *Note that*, our implementation also includes improvements achieved by using better data structures than the original.

6.1 Juggling Patterns

Siu et al. [8, 11] give juggling patterns generation as an application of stream constraint solving. The St-CSP model describes the possible patterns of juggling moves obeying physical laws, parametrised by the number b of balls being juggled and the maximum

number f of upward force units that can be applied to the balls. For physical reasons, $b \leq f$. This problem has variable symmetries. The test cases therefore contain symmetry breaking constraints, and can demonstrate the efficiency of our improvement.

Table 1 gives the experimental results. Our implementation performs much better than the original solver in both metrics. The reduction in time can be as much as 96% for the case ($b = 4, f = 6$), which also achieves a 90% reduction in automaton size.

Table 1: Results for the Juggling Test Cases

	Original		**New**	
Test Case	**Time (s)**	**# of States**	**Time (s)**	**# of States**
(b=4, f=4)	0.00	**5**	**0.00**	**5**
(b=4, f=5)	2.10	481	**0.12**	**121**
(b=4, f=6)	36.28	3601	**1.27**	**361**
(b=5, f=5)	0.10	**6**	**0.01**	**6**
(b=5, f=6)	238.41	3601	**11.33**	**721**
(b=6, f=6)	2.10	**7**	**0.23**	**7**

6.2 Document Circulation Planning Problem

We use the document circulation example in Sect. 5.2 as a class of test cases. The initial condition described is also included in order to demonstrate our improvement. The number n of individuals are varied to generate multiple test cases.

Table 2 gives the experimental results. It shows a significant reduction in both the solving times and the sizes of the solution automata in all test cases. The reduction in search time is at least 64% when $n = 10$ and can be as much as 97% when $n = 13$. As for the sizes of the solution automata, the reduction is at least 53% when $n = 10$ and can achieve 67% when $n = 13$. The results for $n = 14$ are incomparable since the original implementation [8, 11] failed to solve the test case within 1 hour.

Table 2: Results for the Document Circulation Test Cases

	Original Implementation		**New Implementation**	
Test Case	**Time (s)**	**# of States**	**Time (s)**	**# of States**
$n = 10$	5.07	4093	**1.81**	**1920**
$n = 11$	24.23	10236	**4.57**	**3968**
$n = 12$	102.43	20476	**11.50**	**7936**
$n = 13$	1132.53	49147	**28.38**	**16128**
$n = 14$	-	-	**66.59**	**32256**

7 Concluding Remarks

Our contributions in this paper are four-fold. First, we present a novel application of St-CSPs in real-time PID control. We believe this is the first application of CSPs in real-time control on real hardware. The technique was applied to two different pieces

of hardware and achieved stable performance as demonstrated in our video recordings. Second, we propose an improvement on the solving technique, which can lead to orders of magnitude reduction in both the search time and the size of the solution automaton. Third, we identify symmetry breaking and the specification of initial conditions in sequential planning problems as good uses of our improvement. Last but not least, we provide empirical evidence of the proposed improvement. All these takes us one step closer to deploying infinite stream constraint programming in practice.

There is ample room for future work. Here are a few possibilities. We can investigate the relationships among model checking [3], μ-calculus [3] and St-CSPs, as they are all related to Büchi automata. We can also extend the constraint language with more temporal operators, for example `asa` (as soon as), `whenever` and `upon` from the Lucid language [12] to increase our framework's expressiveness. In addition, the link between standard CSPs and St-CSPs can be explored. Standard CSP solving techniques may also help with solving St-CSPs.

Acknowledgements We are grateful to the kind comments and suggestions by the anonymous referees. We thank Simon Wong for building the self-balancing tray and inverted pendulum used for our demonstrations, and Kin-Hong Wong for his help with PID control and robotics. Last but not least, the second author is indebted to Bill Wadge for teaching him dataflow programming and Lucid 25 years ago.

References

1. Bentley, J.: Programming Pearls. Addison-Wesley (2000)
2. Büchi, J.: 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)
3. Clarke, Jr., E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge, MA, USA (1999)
4. Cohen, D., Jeavons, P., Jefferson, C., Petrie, K.E., Smith, B.M.: Constraint symmetry and solution symmetry. In: Proc. AAAI'06. pp. 1589–1592 (2006)
5. Frisch, A.M., Hnich, B., Kiziltan, Z., Miguel, I., Walsh, T.: Global constraints for lexicographic orderings. In: Proc. CP'02. pp. 93–108 (2002)
6. Ghallab, M., Nau, D., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc. (2004)
7. Harvey, W.: Symmetry breaking and the social golfer problem. In: Proc. SymCon'01 (2001)
8. 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)
9. Law, Y.C., Lee, J.H.M.: Symmetry breaking constraints for value symmetries in constraint satisfaction. CONSTRAINTS 11(2-3), 221–267 (2006)
10. Minorsky, N.: Directional stability of automatically steered bodies. Journal of the American Society for Naval Engineers 34(2), 280–309 (1922)
11. Siu, C.F.K.: Constraint Programming on Infinite Data Streams. Ph.D. thesis, Department of Computer Science and Engineering, The Chinese University of Hong Kong (2012)
12. Wadge, W.W., Ashcroft, E.A.: LUCID, the Dataflow Programming Language. Academic Press Professional, Inc. (1985)
13. Winskel, G.: The Formal Semantics of Programming Languages : An Introduction. MIT Press, Cambridge, MA, USA (1993)