Process Calculi for Adaptive Enumeration Strategies in Constraint Programming

Eric Monfroy[1,2], Carlos Olarte[3], and Camilo Rueda[4]

[1] Universidad Técnica Federico Santa María, Valparaíso, Chile
Eric.Monfroy@inf.utfsm.cl
[2] LINA, Université de Nantes, France Eric.Monfroy@univ-nantes.fr
[3] LIX, École Polytechnique Paris, France carlos.olarte@lix.polytechnique.fr
[4] Pontificia Universidad Javeriana, Cali, Colombia crueda@cic.puj.edu.co

Abstract. Constraint programming (CP) has been extensively used to solve a wide variety of problems. Solving a constraint problem mainly consists in two phases: propagation to prune the search space, and enumeration to choose a variable and one of its values for branching. Enumeration strategies are crucial for resolution performances. We propose a framework to model adaptive enumeration strategies using a stochastic, non-deterministic timed concurrent constraint calculus. Using the reactivity of the calculus we can design enumeration strategies that adapt themselves according to information issued from the resolution process and from external solvers such as an incomplete solver. The experimental results show the effectiveness of our approach.

1 Introduction

Constraint Satisfaction Problems (CSP) appears nowadays as a very convenient way to model various industrial applications (e.g. scheduling, timetabling, Boolean satisfiability, etc.). CSP are represented by a set of constraints (relations) between variables. Each variable is associated to a domain which represents the values the variable could be given. The resolution process consists in assigning to each variable a value from its domain such that the constraints are satisfied.

Constraint propagation based solvers are one of the most common methods for solving CSPs. This technique is complete by interleaving enumerations and constraint propagations. Constraint propagation prunes the search tree by eliminating values that cannot participate in any solution. Enumeration creates one branch of the search tree by instantiating a variable ($x = v$) and another branch ($x \neq v$) for backtracking when the first branch does not contain any solution. Although all enumeration strategies that preserve solution sets are valid, they have drastic impacts on efficiency (up to several orders of magnitude). Moreover, no strategy is the best for all problems. The issue is thus to select the right value of the right variable for enumeration. This problem can be tackled with various approaches: static and generic strategies, problem specific strategies [4], dynamic [6] or adaptive strategies to predict [3] or repair strategies [2, 5].

We are interested in modeling adaptive strategies (both for repairing and predicting) using a stochastic, non-deterministic concurrent constraint process calculus, the `sntcc` calculus [12], an extension of `ntcc`[11]. The advantages of using `sntcc` are the following. First, we obtain a clear, formal, and homogeneous (using templates) design of the strategies. Then, the reactive aspect of `sntcc` is used to describe the dynamicity of the strategies, i.e. the ability of adapting, changing, or improving a strategy during the solving process. Additionally, reactivity allows us to introduce on-line expert-users or external solvers knowledge to guide the search. The stochastic aspects of the calculus enables us to rank the strategies with some probabilities of being applied: the higher the probability, the higher the strategy is judged to be efficient. The non-determinism of `sntcc` is used to tie-break strategies that have equal probabilities; we thus introduce randomization from which the solving process can benefit (e.g. [7]).

Using `sntcc` we show several types of adaptive strategies: strategies to diversify enumeration; strategies to reward "good" enumeration strategies (increasing their probabilities of being applied) (see [5]); strategies that use an incomplete but very fast solver (e.g. local search) to identify promising branches leading to a hybrid resolution mechanism; and finally strategies to non-deterministically choose a variable when several variables have the same domain size (see [7]).

This mechanism has been implemented in the Oz language [9] running a simulator of `sntcc`. The experimental results we obtained are more than promising. The adaptive dynamic strategies behave better than the fixed strategies.

The rest of the paper is organized as follows. In Section 2 we give an overview about constraint programming and related work regarding dynamic enumeration strategies. Section 3 presents the `sntcc` calculus a non-deterministic and stochastic extension of `tcc` [14]. Section 4 describes our framework, a generic process for describing strategies, and four instantiations of the framework. Section 5 shows some experimentations and Section 6 concludes the paper and gives some research directions.

2 Constraint propagation-based solvers

In constraint programming (CP), problems are modeled by means of a set V of variables, a set of domains for these variables D and a set of constraints C over the variables. The idea is to choose a value from D for each variable in V s.t. all the constraints are satisfied. Solvers based on constraint propagation alternate pruning of the search tree and split of the search space. The former prunes variable domains by eliminating values that cannot be part of a solution. Nevertheless, propagation is not a complete mechanism. In some cases, the propagation phase never can find a solution nor determine if there are none. In this case a split phase is required, creating two or more subproblems to continue the search. Searching for a solution in CP leads to a search tree where each node represents a new subproblem.

Although all enumeration strategies preserving solutions are valid, they drastically affect the time required to find solutions by several orders of magnitude (see, e.g. [5] for some examples). Thus it is crucial to select a good one (that unfortunately cannot be predicted in the general case) or to eliminate a bad one (by observing and evaluating its behavior). Numerous works were conducted about split strategies. Some studies focused on generic criteria (e.g. minimum domain) for variable selection, and for value selection (e.g. lower bound, or bisection). Some works define strategies for some specific problems (e.g. [4]) where the "best" strategy can be determined according to some static criteria. However, an a priori decision concerning a good variable and value selection is very difficult (and almost impossible in the general case) since strategy effects are rather unpredictable. Information issued from the solving process can also be used to determine the strategy (e.g. [3]). [2] proposes adaptive constraint satisfaction: algorithms that behaves poorly are dynamically changed by the next candidate in a list. In [7], when several choices are ranked equally by the fixed enumeration strategy, randomisation is applied for tie-breaking. In [5], by observing and evaluating the solving process, bad strategies are eliminated whereas good ones are given more chance to be applied.

3 `sntcc` Calculus

Process calculi are mathematical formalisms to model and verify concurrent systems. With (few) operators, they express a wide variety of behaviors such as mobility [13], time and reactivity [14, 11], stochastic and probabilistic choices [8] among others. We are here interested in calculus derived from the Concurrent Constraint model (cc) [15]. cc is based on the concept of constraint as an entity carrying partial information, i.e. conditions on the values variables can take. This model has been extended with the notion of time in `tcc` ([14]), non-determinism and asynchrony in `ntcc` ([11]) and stochastic and probabilistic behavior in `pcc` and `sntcc` ([8, 12]).

In cc, process interactions are determined by the constraints accumulated in a global store (i.e. a set of variables and a conjunction of constraints). The store is used by processes to share information and for synchronization purposes. The store is monotonically refined by adding information using *tell* operations. Additionally, it is possible to test if a constraint c can be entailed from the store by means of so-called *ask* operations. Ask processes allows for synchronisation since they remain blocked until more information is available to entail the query.

`sntcc`[12] extends `ntcc` to model stochastic behavior. In both of them, time is conceptually divided into *time-units*. In a particular time-unit, a deterministic concurrent constraint process receives a stimulus (a constraint) from the environment and it is executed with this stimulus as the initial store. When no further evolution is possible, it responds to the environment with the resulting store. This also determines a residual process which is then executed in the next time-unit (see **next** operator below).

The model of stimulus and responses in `tcc` languages allows to model reactive systems in which agents are in permanent interaction with the environment. In our particular case, the environment will be the processes solving a CSP and stimulus will be statistics taken from this solving process (e.g. number of variable instantiated, depth in the search tree, etc). In turns, strategies modeled with the calculus will respond to the environment with a possible enumeration strategy according to the previous knowledge and rules defined in the strategy.

3.1 Syntax and Processes Description

In this section we present the syntax of the `sntcc` calculus and the intuitive description of the constructs. See [12] for a full treatment of the operational semantics and the logic associated to the calculus.
Process in `sntcc` are built from the following grammar:

$$
\begin{aligned}
P, Q = \mathbf{tell}\ c \mid\ & \sum_{i \in I} \mathbf{when}\ c_i\ \mathbf{do}\ P_i \mid P \| Q \mid \mathbf{local}\ x\ \mathbf{in}\ P \\
& \mid \mathbf{next}\ P \mid \mathbf{unless}\ c\ \mathbf{next}\ P \mid\ !P \mid\ P +_\rho Q \mid\ p(\boldsymbol{x})
\end{aligned}
$$

Process **tell** c adds the constraint c to the current store, thus making c available to other processes in the current time interval. Process $\sum_{j \in J} \mathbf{when}\ c_j\ \mathbf{do}\ P_j$ where J is a finite set of indexes, represents a process that non-deterministically choose a process P_j s.t c_j is entailed by the current store. The chosen alternative, if any, precludes the others. If no choice is possible in the current time unit, all the alternatives are precluded from execution. We shall use $\sum_{j \in J} P_j$ when the guards are **true** ("blind-choice") and we omit "$\sum_{j \in J}$" when J is a singleton. Process $P \| Q$ represents the parallel composition of P and Q. Process **local** x **in** P behaves like P except that the variable x is local in P, i.e., the environment cannot see the information that P produces on x.

The only move of **next** P is a unit-delay for the activation of P. We use $\mathbf{next}^n(P)$ as an abbreviation for $\mathbf{next}(\mathbf{next}(\ldots(\mathbf{next}\, P)\ldots))$, where **next** is repeated n times. **unless** c **next** P executes the process P in the next time-unit if the current store cannot entail the constraint c.

$!P$ stands for the replication of P, i.e. the execution of a copy of P in each time-unit. This construct is equivalent to $P \| \mathbf{next}\ P \| \mathbf{next}\ {}^2 P ....$

$P +_\rho Q$ introduces stochastic behavior. This process evolves to P with a probability ρ and to Q with a probability $1 - \rho$. This probabilistic choice can be extended to multiple processes of the form $P_1 +_{\rho_1} ... +_{\rho_{n-1}} P_n$ normalizing the probabilities to guarantee that $\sum p_i = 1$

Recursion in `sntcc` is defined by means of *processes definitions* of the form

$$p(x_1, .., x_n) \stackrel{\mathtt{def}}{=} A$$

$p(y_1, ..., y_n)$ is an *invocation* and intuitively the body of the process definition (i.e. A) is executed replacing the formal parameter $\boldsymbol{x}$ by the actual parameters $\boldsymbol{y}$. When $|\boldsymbol{x}| = 0$, we shall omit the parenthesis.

To avoid non-terminating sequences of internal reductions (i.e., non-terminating computation within a time interval), recursive calls must be guarded in the context of **next** (see [11] for further details).

4 A framework for modelling adaptive strategies

Devising a framework for dynamic enumeration strategies resembles the idea of reactive systems [1]. Resolution processes can be viewed as the environment adding new information (stimulus) to the system to produce possibly "better" enumeration strategies. Here we propose modelling reactive and adaptive enumeration strategies using `sntcc`. Stimulus will be statistics taken from the solvers (e.g. depth of the search tree, number of variable instantiated, etc.) and the response of the system (resulting store in each time-unit) will be the variable (var) and the value (val) for enumeration. Constructs in the calculus will help us to define rules to choose the "*best*" strategy in each node of the search tree.

Defining dynamic enumeration strategies as `sntcc` processes has several advantages, namely 1) The stimulus-response model allows to clearly define the interaction between the resolution process and the dynamic enumeration strategies. 2) Complex synchronization patterns can be defined (due to blocking asks) allowing us to coordinate different processes adding new information to guide the search. In particular, we show how we can implement enumeration strategies guided by external solvers such as local search. In this case the framework can be viewed as a solver coordinator. 3) The stochastic component of the calculus allows us to rank strategies with some probabilities of being applied. Additionally, probabilities of each strategy can be dynamically changed during the solving process. 4) Non-determinism can be used to tie-break strategies that are equally ranked. Finally, 5) the parallel operator allows us to integrate incrementally new processes telling information useful to adapt the strategy or combine rules in a compositional way.

Given a CSP P with a set of constraints C and a dynamic strategy S (i.e a `sntcc` process) the framework works as follows: 1) The constraint solver creates the first node of the search tree imposing the constraints in C. 2) The enumeration procedure creates a new `sntcc` time-unit. When doing that, it feeds in the `sntcc` store with statistical information of the solving process as stimulus. 3) A `sntcc` interpreter computes the resulting store w.r.t. the stimulus and the strategy S. 4) After stability in the `sntcc` store, the variable (var) that must be split and the value (val) that it must take are entailed from the `sntcc` store. 5) Finally the enumeration procedure creates as usual two choices in the constraint solver: $var = val$ and $var \neq val$ leading to a new phase of propagation. In this way, the search tree is explored using dynamic choices according to S and statistics taken from the resolution process.[5]

A template of a strategy S in our framework can be defined as the parallel composition of three processes:

$$\textbf{Strategy} = \textbf{Choice} || \textbf{Update\_Probabilities} || \textbf{External\_knowledge}$$

[5] `sntcc` constraints and store must not be confused with constraints C in the CSP. The former are used to determine the evolution of the strategy and the latter define the problem to be solved

Update_Probabilities (**U_P**) changes the probability of each strategy according to user defined rules. Expert-user knowledge or results taken from external solvers (i.e. local search, genetic algorithms, etc.) can be used to adapt the strategy during its execution by defining an **External_knowledge** (**E_K**) process. Additionally, one could add on-line new information (constraints) as stimulus to fix some parameters in the strategy. This can be useful when expert users observing the solving process can give some hints for improving the search. Finally, **Choice** makes a probabilistic or non-deterministic choice according to information inferred from both **U_P** and **E_K**. In the following, we instantiate this generic process to define some strategies that will be tested in Section 5.

4.1 Diversifying Enumeration Strategies

Our first dynamic strategy consists in applying alternatively three different static value selection: *min* (selects the lower bound of the domain), *max* (the upper bound) and *mid* (the closest element to the middle of the domain) leading to a diversification of the value selection. The variable to be split is always the variable with the smallest domain (i.e the variable selection is fixed). The behavior of this strategy can be captured by the following process:

$$\begin{aligned} \mathbf{Choice} \equiv \quad & \mathbf{tell}\ (val = min) \parallel \mathbf{next\ tell}\ (val = mid) \parallel \\ & \mathbf{next}^2\ \mathbf{tell}\ (val = max) \parallel \mathbf{next}^3\ \mathbf{Choice} \\ \mathbf{S_1} \equiv \mathbf{Choice} & \end{aligned}$$

4.2 Probabilistic Choice of the Value Selection

Although S_1 dynamically changes the value selection, it always applies the same pattern of choices. The following strategy changes the probability of applying a value selection according to the results obtained previously. The number of variables that has not been instantiated yet is the stimulus introduced by the constraint solver. The idea is to increase the probability of some value selection when it allows good pruning in the next node of the search tree. If the selection leads to a failed node in the search tree, its probability is decreased giving more chance to other strategies to be applied.

$$\begin{aligned} \mathbf{Init} \ & = \mathbf{tell}\ (\rho Min = \tfrac{1}{3} \wedge \rho Mid = \tfrac{1}{3} \wedge \rho Max = \tfrac{1}{3}) \\ \mathbf{U\_P} \ & = \mathbf{when}\ VarNI < VarNI'\ \mathbf{do}\ increase(val') + \\ & \quad\ \mathbf{when}\ VarNI \geq VarNI'\ \mathbf{do}\ decrease(val') \\ \mathbf{Choice} & = \mathbf{tell}\ (val = min)\ +_{\rho Min}\ \mathbf{tell}\ (val = mid)\ +_{\rho Mid}\ \mathbf{tell}\ (val = max) \\ \mathbf{S_2} \ & = \mathbf{Init} \parallel \mathbf{!Choice} \parallel \mathbf{!next}\ (\mathbf{U\_P}\ )) \end{aligned}$$

In the expression above, $VarNI$ (resp. $VarNI'$) represents the number of variables not instantiated in the current (resp. previous) node of the search tree. ρMin, ρMid and ρMax are the probabilities of choosing each value selection. *increase* (resp. *decrease*) increases (resp. decreases) in a constant factor the probability of the previous value selection (val') updating the rest of probabilities to guarantee $\rho Min + \rho Mid + \rho Max = 1$. Once **U_P** has determined the new probabilities, **Choice** makes a probabilistic choice between each alternative. Thus, the behavior of this strategy is to apply (probabilistically) the value selection that has not failed lately.

4.3 Local search based decisions

Integrating complete and incomplete methods for constraint solving has been shown promising for reducing the time required to obtain solutions in CP [10]. In our case, we use a descent algorithm for *local search* to determine the "best" value selection to continue the search in the CP solver. Let $V = \{v_1, ..., v_n\}$ be the set of variables of the CSP with domains $D = \{d_1, ..., d_n\}$. The process $LS(v_i, value, cv)$ computes local search with a neighborhood function $NB : D \rightarrow 2^D$ s.t. for all configurations c and for all $c_i \in NB(c)$, $c_i \downarrow_{v_i} = value$ i.e, the assignment $v_i = value$ is preserved during the search. In our tests, NB computes a set of configurations by changing one variable not yet determined (i.e. $|dom(v_i)| > 1$) at a time. Additionally, when no moves improve the current configuration, a random restart is performed introducing diversification in the algorithm. After a fixed number of iterations, the LS process binds cv to the number of constraints violated in the best solution found. In this way, synchronization with the rest of the system is achieved. Thus, LS processes such as $LS(v_i, min(v_i), ls_{min})$, $LS(v_i, mid(v_i), ls_{mid})$ and $LS(v_i, max(v_i), ls_{max})$ can be executed in parallel to determine the most promising branch to search for a solution (*min*, *mid*, and *max* being the usual value selections).

Since computing local search in each time-unit (node in the search tree) may cause overloads, we can control when the strategy uses LS-based decision. For example, we can apply LS only in the 3 first levels of the search tree (i.e $depth \leq 3$) or when more than a certain number of variables are instantiated. This kind of control can be achieved modifying the predicate *cond* in S_3:

$$\begin{array}{ll}
\textbf{E\_K} & = LS(v_i, min(v_i), ls_{min}) || LS(v_i, mid(v_i), ls_{mid}) || LS(v_i, max(v_i), ls_{max}) \\
\textbf{Choose} & = \textbf{when } ls_{min} \leq ls_{mid} \wedge ls_{min} \leq ls_{max} \textbf{ do tell } (val = min) \\
 & \quad + \textbf{when } ls_{mid} \leq ls_{min} \wedge ls_{mid} \leq ls_{max} \textbf{ do tell } (val = mid) \\
 & \quad + \textbf{when } ls_{max} \leq ls_{min} \wedge ls_{max} \leq ls_{mid} \textbf{ do tell } (val = max) \\
\mathbf{S_3} & = !(\textbf{when } cond \textbf{ do } (\textbf{local } ls_{min}, ls_{mid}, ls_{max} \textbf{ in E\_K} || \textbf{Choose}) + \\
 & \quad \textbf{when } \neg cond \textbf{ do } S_{1/2})
\end{array}$$

Note that when S_3 does not apply local search based-decisions, it behaves like S_1 or S_2.

S_3 can be improved by evaluating more than one variable with LS to find the most promising branch. This can be done extending **E_K** and **Choose** processes as follows:

$$\begin{array}{ll}
\textbf{E\_K} & = \Pi_{vi \in V_{Selected}} (LS(v_i, min(v_i), ls_{min_i}) || LS(v_i, mid(v_i), ls_{mid_i}) || \\
 & \qquad LS(v_i, max(v_i), ls_{max_i})) \\
\textbf{Choose} & = \textbf{local } MinCost \textbf{ in tell } MinCost = Min(ls_{min_{1..n}}, ls_{mid_{1..n}}, ls_{max_{1..n}}) || \\
 & \sum_{i \in 1..n} (\textbf{when } ls_{min_i} = MinCost \textbf{ do tell } (var = v_i \wedge val = min) + \\
 & \qquad \textbf{when } ls_{mid_i} = MinCost \textbf{ do tell } (var = v_i \wedge val = mid)) \\
 & \qquad \textbf{when } ls_{max_i} = MinCost \textbf{ do tell } (var = v_i \wedge val = max))
\end{array}$$

where Π stands for parallel composition of several processes and $V_{Selected}$ denotes the set of variables to be evaluated. **E_K** thus launches several LS processes that are synchronized with the non-deterministic choice in **Choose**. In this way, the variable and the value for that variable with minimum cost w.r.t. to the local search algorithms is chosen.

4.4 Randomized Variable Selection

This strategy aims at randomly (non-deterministically) choosing a variable when multiple variables have the same domain size. Hence, we are able to tie-break strategies with the same probability of being applied. This mechanism has shown good performances for some CSPs in [7].

To implement this strategy, the state of the variables in the solver (i.e. their domains) must be sent as a stimulus. In the `sntcc` store we thus have "a copy" of the variables of the solver. This strategy computes the minimal size (*min_size*) of the variables domains to filter the variables whose size is equal to *min_size*. This new list of variables is stored in a local variable $VSet$. Next, the strategy chooses non-deterministically one variable in the set $VSet$:

$$\begin{aligned}
\mathbf{U\_P} &= \mathbf{tell}\ (Min = min\_domain(Problem\_vars)\ ) \,||\\
&\ \mathbf{tell}\ (VSet = filter(Problem\_vars, Min)\ )\\
\mathbf{Choose} &= \textstyle\sum_{v_i \in VSet} \mathbf{tell}\ (Var = v_i)\\
\mathbf{S_4} &= !\ \mathbf{local}\ Min, VSet\ \mathbf{in}\ (\mathbf{Choose} \,||\, \mathbf{Update})
\end{aligned}$$

Note that S_4 only performs variable selection. Executing it in parallel with some of the previously defined strategies leads to a scheme of dynamic variable and value selection.

5 Experimental Results

Tests presented in this section were performed in a Pentium 4 1.80GHz CPU running Linux Gentoo, kernel 2.6.15 and Oz system 1.3.1.

We run different instances of the canonical problems *Magic Sequence*, *Knights*, *Magic Square* and *N-Queens* (see [9] for an Oz program solving these problems). For each problem we compare static strategies with dynamic strategies in Tables 1, 2, 3 and 4 [6] respectively. We measure the average number of nodes explored (including failed nodes) and the average time to find one solution. In what follows, we present a brief description of each problem and the results we obtained.

Magic Square: The Magic Square Puzzle consists in finding a $N \times N$ matrix such that: 1) Every field of the matrix is an integer between 1 and N^2. 2) The fields of the matrix are pairwise distinct. 3) The sums of the rows, columns, and the two main diagonals are all equal.

[6] "-" denotes more than 1.000.000 nodes.

For this problem we compare the static value selections *min*, *mid* and *max* provided by Oz selecting always the variable with the smallest domain. Strategy S_3 executes S_1 after applying LS-based decision in the first 3 levels of the search tree and evaluates the two variables with shortest domain in each case.

Note that for instances where $N \geq 4$, dynamic strategies outperform static value selections. Furthermore, when the difference between the number of nodes explored is not significative (i.e $N < 4$), execution time is better with the static strategies due to the overload of the execution of the `sntcc` interpreter.

Knights: The idea is to find a sequence of knights moves on a $N \times N$ chessboard such that each field is visited exactly once and that the moves return the knight to the starting field, which is fixed to the lower left field.

For this problem we compare also static value selection *min*, *mid* and *max* vs dynamic strategies S_1, S_2. In this case S_1 and S_2 slightly outperform Min regarding the number of nodes explored and show better performances w.r.t. Max and Mid.

Magic Sequence: A magic sequence of length n is a sequence $x_0, x_1, \ldots x_{n_1}$ of integers such that for every $0 \leq i < n-1$: 1) x_i is an integer between 0 and $n-1$ and 2) the number i occurs exactly x_i times in the sequence.

Similar to the *Knights* problem, we compare the same strategies. According to the form of solution for this problem where the first value in the sequence must be a value close to the maximum (representing the number of zeros in the sequence) and the next one must be a small one, S_1 turns out to be a good strategy solving all the instances in 4 nodes. As expected, if it starts selecting the minimum or the mid value, we obtain results similar to the static strategies.

N-Queens: This problem consists in placing N queens on an $N \times N$ chess board such that no two queens attack each other.

We compare S_4 with the following fixed strategies: *naive* (selects the leftmost variable), *size* (selects the leftmost variable whose domain is minimal), *min* (the leftmost variable whose lower bound is minimal) and *max* (the leftmost variable whose upper bound is maximal). In all the cases the value selected is the closest to the middle of the domain (*mid*). Note that the *size* strategy is known for this problem as a good strategy exploring a few amount of nodes before finding a solution. S_4 makes some improvements randomizing the variable selection when multiple variables have the same domain size. Notice also that *size* is better regarding the execution time due to the overload of the `sntcc` interpreter in S_4.

6 Concluding remarks

We proposed a generic framework to model and integrate adaptive enumeration strategies in a constraint propagation-based solver. This framework is based on `sntcc`, a stochastic non-deterministic concurrent constraint calculus. The reactivity of the calculus allows the strategies to adapt themselves according to observations of the resolution. Additionally, parallel composition enables us to add new processes to guide the search, even external processes such as incomplete

	Max		Mid		Min		S_1		S_2		S_3	
N	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
3	16	0.10	6	0.09	27	0.09	9	0.05	6.05	0.35	8.67	0.49
4	1209	0.13	380	0.16	480	0.19	3449	19.45	1611.1	8.16	101.56	1.70
5	-	-	151211	11.11	158006	10.4	573	3.10	489.10	2.51	387.5	4.01
6	-	-	-	-			2929	15.60	1585.38	8.04	528.27	6.93
7	-	-	-	-			2026	11.03	2783.87	14.06	203.25	14.62
8	-	-	-	-			257	1.77	2267.27	11.90	495.55	13.33

Table 1. Magic Square Problem

	Max		Mid		Min		S_1		S_2	
N	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
8	34	0.43	46	0.40	44	0.39	33	0.82	35.07	0.84
10	1838	2.41	69	0.43	76	0.44	61	1.26	69.71	1.36
12	98	0.56	-	-	117	0.56	95	1.83	110.5	2.06
14	1116	3.09	-	-	174	0.81	142	2.83	146.63	2.9

Table 2. Knights Problem

methods (e.g. local search) or heuristics from expert-users. The framework was instantiated with 4 dynamic strategies that showed better performances than static strategies to solve four well known CSPs.

The main advantage in using this framework is that one can express complex enumeration strategies only adding/modifying *sntcc* processes in the strategy definition. Issues concerning synchronization, choices, etc. are relayed to the formal behavior provided by the calculus. Furthermore combining different enumeration strategies can be easily achieved by using parallel composition or probabilistic/non-deterministic choices. In this way, strategies can be improved running them in parallel. Finally, an interesting feature is that external solvers can be synchronized with the constraint-based solving process. Thus, information provided by them can guide the variable/value selection during the search.

Our interest in this framework is twofold: 1) to integrate more indicators from the solving process to better guide the search, and 2) to integrate more external solvers and achieve their coordination using process calculi. The final idea is to provide a generic and adaptive framework for hybrid resolution of CSPs.

References

1. G. Berry and G. Gonthier. The esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 1992.
2. J. Borrett, E. Tsang, and N. Walsh. Adaptive constraint satisfaction: The quickest first principle. In *Proc. of ECAI'1996*, pages 160–164. John Wiley and Sons, 1996.
3. T. Carchrae and J. C. Beck. Low-Knowledge Algorithm Control. In *Proceedings of the National Conference on Artificial Intelligence, AAAI 2004*, pages 49–54, 2004.
4. Y. Caseau and F. Laburthe. Improved clp scheduling with task intervals. In *Proceedings of ICLP'1994*, pages 369–383. MIT Press, 1994.

	Max		Mid		Min		S_1		S_2	
N	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
70	31	0.41	35	0.39	35	0.39	4	0.42	35	0.83
90	41	0.48	45	0.41	45	0.40	4	0.42	45	0.99
110	51	0.58	55	0.42	55	0.42	4	0.42	55	1.18
130	61	0.74	65	0.47	65	0.45	4	0.45	65	1.41
150	71	0.98	75	0.52	75	0.51	4	0.45	75	1.64

Table 3. Magic Sequence Problem

	$Naive$		$Size$		Max		Min		S_4	
N	Nodes	Time(s)	Nodes	Time	Nodes	Time	Nodes	Time	Nodes	Time
32	119636	4.1	79	0.32	-	-	-	-	73.15	0.49
64	-	-	89	0.4	-	-	-	-	71.7	0.67
128	-	-	127	0.42	-	-	-	-	126.45	1.36
256	-	-	257	0.45	-	-	-	-	254.3	4.01

Table 4. N-queens Problem

5. C. Castro, E. Monfroy, C. Figueroa, and R. Meneses. An approach for dynamic split strategies in constraint solving. In *Proceedings of MICAI'05*, volume 3789 of *LNCS*, pages 162–174. Springer, 2005.
6. P. Flener, B. Hnich, and Z. Kiziltan. A meta-heuristic for subset problems. In *Proceedings of PADL'2001*, volume 1990 of *LNCS*, pages 274–287. Springer, 2001.
7. C. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of AAAI'98*, pages 431–437, Madison, Wisconsin, 1998.
8. V. Gupta, R. Jagadeesan, and V. A. Saraswat. Probabilistic concurrent constraint programming. In *Proc. of Int. Conf. on Concurrency Theory*, pages 243–257, 1997.
9. S. Haridi and N. Franzn. *Tutorial of Oz.*, 2004. Available at `www.mozart-oz.org`.
10. E. Monfroy, F. Saubion, and T. Lambert. Hybrid csp solving. In *Proceeding of FroCoS'05*, volume 3717 of *LNCS*, pages 138–167. Springer, 2005. invited paper.
11. M. Nielsen, C. Palamidessi, and F. D. Valencia. Temporal concurrent constraint programming: Denotation, logic and applications. In *Special Issue of Selected Papers from EXPRESS'01, Nordic Journal of Computing*, 2001.
12. C. Olarte and C. Rueda. A stochastic non-deterministic temporal concurrent constraint calculus. In *Proceedings of SCCC'05*, 2005.
13. J. P. R. Milner and D. Walker. A calculus of mobile processes, Parts I and II. *Journal of Information and Computation*, 100:1–77, Sept. 1992.
14. V. Saraswat, R. Jagadeesan, and V. Gupta. Fundation of timed concurrent constraint programming. In *IEEE Symp. on Logic in Comp. Science*. IEEE, 1994.
15. V. Saraswat, M. Rinard, and P. Panangaden. Semantic foundations of concurrent constraint programming. In *Proc. of POPL'91*, pages 333–353. ACM Press, 1991.